X/Open CAE Specification

IPC Mechanisms for SMB

X/Open Company Ltd.

X/Open CAE Specification

IPC Mechanisms for SMB

ISBN: 1 872630 28 6
X/Open Document Number: XO/CAE/91/500

Published by X/Open Company Ltd., U.K.

Any comments relating to the material contained in this document may be submitted to X/Open at:

X/Open Company Limited
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

XoSpecs@xopen.org

Contents

List of Tables

Preface

X/Open

X/Open is an independent, worldwide, open systems organisation supported by most of the world's largest information systems suppliers, user organisations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

X/Open's strategy for achieving this goal is to combine existing and emerging standards into a comprehensive, integrated, high-value and usable open system environment, called the Common Applications Environment (CAE). This environment covers the standards, above the hardware level, that are needed to support open systems. It provides for portability and interoperability of applications, and so protects investment in existing software while enabling additions and enhancements. It also allows users to move between systems with a minimum of retraining.

X/Open defines this CAE in a set of specifications which include an evolving portfolio of application programming interfaces (APIs) which significantly enhance portability of application programs at the source code level, along with definitions of and references to protocols and protocol profiles which significantly enhance the interoperability of applications and systems.

The X/Open CAE is implemented in real products and recognised by a distinctive trade mark — the X/Open brand — that is licensed by X/Open and may be used on products which have demonstrated their conformance.

X/Open Technical Publications

X/Open publishes a wide range of technical literature, the main part of which is focussed on specification development, but which also includes Guides, Snapshots, Technical Studies, Branding/Testing documents, industry surveys, and business titles.

There are two types of X/Open specification:

- *CAE Specifications*

  CAE (Common Applications Environment) specifications are the stable specifications that form the basis for X/Open-branded products. These specifications are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement an X/Open CAE specification can enjoy the benefits of a single, widely supported standard. In addition, they can demonstrate compliance with the majority of X/Open CAE specifications once these specifications are referenced in an X/Open component or profile definition and included in the X/Open branding programme.

  CAE specifications are published as soon as they are developed, not published to coincide with the launch of a particular X/Open brand. By making its specifications available in this way, X/Open makes it possible for conformant products to be developed as soon as is practicable, so enhancing the value of the X/Open brand as a procurement aid to users.

- *Preliminary Specifications*

  These specifications, which often address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations, are released in a controlled manner for the purpose of validation through implementation of products. A Preliminary specification is not a draft specification. In fact, it is as stable as X/Open can make it, and on publication has gone through the same rigorous X/Open development and review procedures as a CAE specification.

  Preliminary specifications are analogous to the *trial-use* standards issued by formal standards organisations, and product development teams are encouraged to develop products on the basis of them. However, because of the nature of the technology that a Preliminary specification is addressing, it may be untried in multiple independent implementations, and may therefore change before being published as a CAE specification. There is always the intent to progress to a corresponding CAE specification, but the ability to do so depends on consensus among X/Open members. In all cases, any resulting CAE specification is made as upwards-compatible as possible. However, complete upwards-compatibility from the Preliminary to the CAE specification cannot be guaranteed.

In addition, X/Open publishes:

- *Guides*

  These provide information that X/Open believes is useful in the evaluation, procurement, development or management of open systems, particularly those that are X/Open-compliant. X/Open Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming X/Open conformance.

- *Technical Studies*

  X/Open Technical Studies present results of analyses performed by X/Open on subjects of interest in areas relevant to X/Open's Technical Programme. They are intended to communicate the findings to the outside world and, where appropriate, stimulate discussion and actions by other bodies and the industry in general.

- *Snapshots*

  These provide a mechanism for X/Open to disseminate information on its current direction and thinking, in advance of possible development of a Specification, Guide or Technical Study. The intention is to stimulate industry debate and prototyping, and solicit feedback. A Snapshot represents the interim results of an X/Open technical activity. Although at the time of its publication, there may be an intention to progress the activity towards publication of a Specification, Guide or Technical Study, X/Open is a consensus organisation, and makes no commitment regarding future development and further publication. Similarly, a Snapshot does not represent any commitment by X/Open members to develop any specific products.

Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision, in this case as the subject technology develops and to align with emerging associated international standards. X/Open makes a distinction between revised specifications which are fully backward compatible and those which are not:

- a new *Version* indicates that this publication includes all the same (unchanged) definitive information from the previous publication of that title, but also includes extensions or additional information. As such, it *replaces* the previous publication.

- a new *Issue* does include changes to the definitive information contained in the previous publication of that title (and may also include extensions or additional information). As such, X/Open maintains *both* the previous and new issue as current publications.

Corrigenda

Most X/Open publications deal with technology at the leading edge of open systems development. Feedback from implementation experience gained from using these publications occasionally uncovers errors or inconsistencies. Significant errors or recommended solutions to reported problems are communicated by means of Corrigenda.

The reader of this document is advised to check periodically if any Corrigenda apply to this publication. This may be done in any one of the following ways:

- anonymous ftp to ftp.xopen.org
- ftpmail (see below)
- reference to the Corrigenda list in the latest X/Open Publications Price List.

To request Corrigenda information using ftpmail, send a message to ftpmail@xopen.org with the following four lines in the body of the message:

```
open
cd pub/Corrigenda
get index
quit
```

This will return the index of publications for which Corrigenda exist. Use the same email address to request a copy of the full corrigendum information following the email instructions.

This Document

This document is a CAE Specification (see above). It is intended for application programmers designing distributed client/server applications whose server component is portable to all X/Open-compliant systems. This specification defines the Application Programming Interfaces to be used and the necessary Server Message Block protocol extensions for interprocess communication.

Chapter 1 describes the structure of the book and explains the relationship of this specification to other X/Open documents.

Readers should be experienced C programmers. Programmers writing applications using the API covered in this document should read Chapters 1 to 3. Programmers writing applications to implement an LMX server should have read the referenced document **Protocols for X/Open PC Interworking: SMB** before reading the whole of this document.

Throughout this book C language conventions are used, for example, a hexadecimal number has the prefix 0x.

Trademarks

Ethernet® is a registered trademark of Xerox Corporation.

IBM® is a registered trademark of International Business Machines Corporation.

LAN Manager™ is a trademark of Microsoft Corporation.

Microsoft®, MS® and MS-DOS® are registered trademarks of Microsoft Corporation.

NFS® is a registered trademark of Sun Microsystems.

OS/2® is a registered trademark of International Business Machines Corporation.

Palatino™ is a trademark of Linotype AG and/or its subsidiaries.

PC-NFS™ is trademark of Sun Microsystems.

Sun Microsystems® is a registered trademark of Sun Microsystems.

X/Open™ and the ''X'' device are trademarks of X/Open Company Ltd. in the U.K. and other countries.

Referenced Documents

(PC)NFS

Protocols for X/Open PC Interworking: (PC)NFS, Developers' Specification, X/Open Company Ltd., 1991 (XO/DEV/90/030).

SMB

Protocols for X/Open PC Interworking: SMB, Developers' Specification, X/Open Company Ltd., 1991 (XO/DEV/91/010).

XPG3

X/Open Portability Guide, Issue 3, Volume 2, XSI System Interface and Headers.

XTI

Revised XTI (X/Open Transport Interface), Developers' Specification, X/Open Company Ltd., 1990 (XO/DEV/90/060), Appendix D, Use of XTI to Access NetBIOS, Preliminary Specification.

The Waite Group OS/2 Programmer's Reference.

Chapter 1

Introduction

X/Open realises how important it is to facilitate interworking between personal computers and X/Open-compliant systems in a standardised way.

With the referenced documents **Protocols for X/Open PC Interworking: (PC)NFS** and **Protocols for X/Open PC Interworking: SMB** methods were defined to access remotely resources that are local to an X/Open-compliant system. Application Programming Interfaces (APIs) are not addressed in these specifications. This document makes it possible to write distributed client/server applications whose server component is portable to all systems that implement this specification. Another method is provided in Appendix D of the referenced document **Revised XTI (X/Open Transport Interface)**. This method makes it possible for a PC application utilising the NetBIOS interface to communicate with its peer application running on an X/Open-compliant system. NetBIOS is an important *de facto* standard interface for networking PCs.

The Server Message Block (SMB) protocol from Microsoft Corporation, as defined in the referenced document **Protocols for X/Open PC Interworking: SMB**, provides file and print-sharing facilities. This specification defines the APIs to be used on the X/Open-compliant system and necessary SMB protocol extensions for Interprocess Communication (IPC).

The X/Open LAN Manager (LMX) IPC mechanisms include three facilities:

mailslots — are named receptors for simple unidirectional communication.

named pipes — let applications exchange information in a connection-oriented fashion in either byte stream or message mode. These are distinct from pipes or first in first out (FIFO) special files as defined in the **X/Open Portability Guide**, **Issue 3**, **Volume 2**, **XSI System Interface and Headers**.

messaging — allows notification messages to be sent to DOS and OS/2 systems running a messaging service.

The service models of the three IPC mechanisms are defined in Chapter 2.

Chapter 3 defines the LMX IPC APIs that are supported on the X/Open-compliant system. These APIs are defined to be as similar as possible to the interfaces defined for DOS and OS/2 by Microsoft's LAN Manager 1.0.

Chapter 4 provides information to the developer of an X/Open-compliant LMX server implementation on mapping the APIs to SMB protocol elements.

Chapter 5 specifies the extended SMB protocol elements necessary to implement the APIs for named pipes and mailslots. These protocol elements are additions to the SMBs for PC file and print sharing as defined in the referenced document **Protocols for X/Open PC Interworking: SMB**. Chapter 5 supersedes Appendix B in the referenced document **Protocols for X/Open PC Interworking: SMB**.

Chapter 6 specifies a distinct set of SMB protocol elements necessary to support messaging; this supersedes Appendix C in the referenced document **Protocols for X/Open PC Interworking: SMB**.

The usage of LMX IPC mechanisms between processes on the same X/Open-compliant system is not defined, but is not precluded.

Chapter 2

SMB IPC Service Model

This chapter describes the basic service model for the mailslot, named pipe and messaging IPC mechanisms. It describes the higher-level operations that must occur (in addition to the API calls specified in Chapter 3) to enable one of these mechanisms between processes. Implementation-defined operations are described in abstract terms.

2.1 Security Overview

In the referenced document **Protocols for X/Open PC Interworking: SMB** the concepts of *user-level* and *share-level* security are discussed. This document should be referenced for additional information, but a short description of these security mechanisms follows.

In LMX, resources are *shared* by making the name of the resource available for access from the network. For example, the server named XOPEN may create a resource name SMB which contains the contents of this document. This allows systems on the network to connect to the resource and access this data. There are two forms of security which can be applied to the data located behind the resource name SMB. One form of security within LMX is to place access control at the level of the resource. In this case, if users on the network know the name of the resource and the password associated with the resource they are able to connect and access all data located on the LMX server identified by the resource name SMB. This is called share-level security.

Another form of security is to assign a user context to anyone making a connection to a resource. This way more control can be granted to people connecting to the same resource location. For example, a user called ARTHUR could be the author of the document and a user called REED could be a reviewer for the document. When user-level security is used the LMX server distinguishes between users when the connection to the resource is made. Now, ARTHUR can have read/write access to the documents and REED is only able to read the data located by the resource.

2.2 Naming

2.2.1 Named Pipe and Mailslot Names

An LMX implementation supports the following hierarchy of names for named pipes and mailslots.

IPC name
resource name
LMX server name

The first layer, the *LMX server name*, is the name used by users of LMX servers to identify the specific LMX server desired. There is no predefined restriction on the length and format of a server name. The term **computername** is used to refer to nodes on the network that are addressable through SMB IPC mechanisms but are not running an LMX server implementation. The LMX server name is converted by clients into *NetBIOS names*, which are described below. Each LMX server supports a collection of *resource names*. A resource name represents a resource to the client of the LMX server. Examples of resources are printers, file storage and IPC. In the context of this specification, the only resource is IPC. The resource name used to support named pipes and mailslots is fixed and is ''IPC$''. This resource name is not visible to applications using named pipes and mailslots, but is known by the LMX client implementation. A resource name follows the Uniform Naming Convention (UNC). UNC names are constructed from names having an 8.3 format that are separated by a backslash (\). An 8.3 format name consists of two components: a one to eight character name must be present and an optional one to three character extension may be added. The two components are separated by a period, hence the term 8.3 format. Legal characters are alphanumerics and punctuation symbols except for the period (.), backslash (\) and slash (/). These names are case-insensitive.

The *IPC name* is supplied by the users of named pipes and mailslots. An IPC name is a UNC name. There are two types of IPC names:

mailslot — These names start with the first 8.3 format component **\MAILSLOT** followed by at least one additional component.

named pipe — These names start with the first 8.3 format component **\PIPE** followed by at least one additional component.

An LMX implementation has a restriction on the length of an IPC name which is at least 128 bytes.

2.2.2 NetBIOS Names

NetBIOS names are used to establish the actual connections or *NetBIOS sessions* between an LMX client and LMX server systems (see the referenced document **Protocols for X/Open PC Interworking: SMB**, Chapters 13, 14 and 15). A NetBIOS name is 16 bytes in length. The 16th byte of a NetBIOS name has particular meaning. If the value of the 16th byte is 0x20, the name represents a name requesting service from an LMX server. Other values for the 16th byte are discussed in Chapter 6.

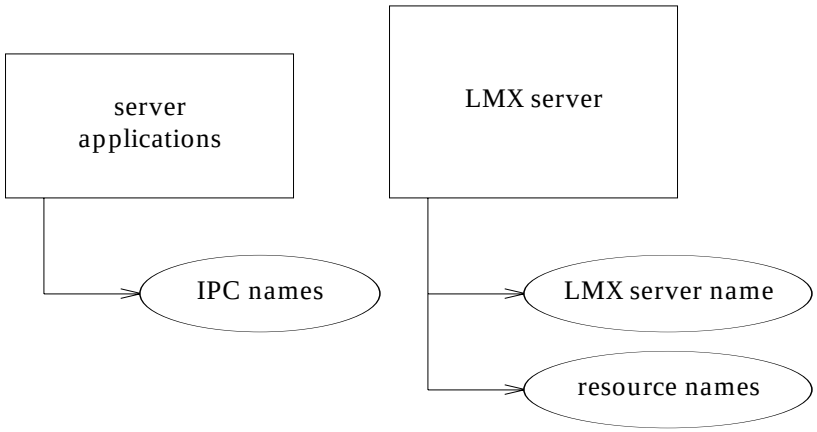
For the purposes of this specification, NetBIOS names are obtained by converting the LMX server name to upper case and padding with 0x20 up to 16 bytes. The 16th byte is always set to 0x20, so the length of the LMX server name is restricted to 15 characters.

2.2.3 Messaging Names

Messaging names are converted to NetBIOS names via the rules described in Chapter 6 on page 119. They consist of only one component representing the individual that should receive the message. A messaging name can be from 1 to 15 characters in length.

2.2.4 Name Ownership

It is important to recognise where the ownership of each of these names resides.



As shown above, an LMX server owns one LMX server name to identify itself, and one or more resource names. For the support of named pipes and mailslots, the resource name ‘‘IPC$’’ must be available. Server applications own one or more IPC names. The IPC name ‘‘**LANMAN**’’ is reserved.

2.3 Mailslot Service Model

2.3.1 Distributed Computing Model

Mailslots provide a unidirectional, connectionless, interprocess communication (IPC) mechanism.

There are two classes of mailslot:

- Class 1 mailslots are guaranteed - the message is delivered or the sender is notified of failure.
- Class 2 mailslots are simply sent; no return code informs the sender of an unsuccessful delivery. Class 2 mailslots can be used to send messages to all systems that have a mailslot created with the same IPC mailslot name.

2.3.2 Naming

The name for a mailslot has the following form:

local:	**\MAILSLOT\mailslotname**
remote:	**\\computername\MAILSLOT\mailslotname**
broadcast:	***\MAILSLOT\mailslotname**

computername identifies the node where the mailslot was created.

***** means broadcast to all nodes; may only be used with a Class 2 mailslot.

mailslotname the name assigned to the mailslot by the creating server application. Names for mailslots are IPC names.

2.3.3 Security

If the specified mailslot does not exist on a Common Applications Environment (CAE) system, the LMX server security and access control mechanisms may not be enforced.

In a CAE it may be desirable to limit which processes can create a mailslot. This can be done by limiting mailslot creation to processes belonging to a special user or group (such as *lmpipe*) or to a process of appropriate privileges. For example, on Class 1 mailslots it is necessary to connect to the ‘‘IPC$’’ share on the server system. Therefore, the access to this share may be restricted to a certain group of users.

2.3.4 Inheritance

A mailslot is not inheritable by a child process.

2.3.5 Process Termination

When the CAE server application that created the mailslot terminates, either via a normal exit condition or a process failure, the LMX server removes the name of the mailslot from the mailslot name space.

2.3.6 API Summary

The following table summarises the APIs for mailslot usage. The first column lists the API; the second column shows whether the API is issued by the sending process or the creating process; and the third column is a short description.

API	Issued by	Description
DosDeleteMailslot	creator	Delete a mailslot.
DosMailslotInfo	creator	Retrieve information about a mailslot.
DosMakeMailslot	creator	Create a mailslot and return its handle.
DosPeekMailslot	creator	Read a message without removing it from the mailslot.
DosReadMailslot	creator	Read the next message from the mailslot.
DosWriteMailslot	sender	Send a message to a mailslot.

2.3.7 Sample Mailslot Communication Scenario

Creator side	Sender side
A process creates the mailslot via *DosMakeMailslot*( ). This call returns the mailslot handle used in future operations on this mailslot.	
	A process calls *DosWriteMailslot*( ), specifying the name of the mailslot.
The process may call *DosPeekMailslot*( ) to see if data has arrived or *DosReadMailslot*( ) to read the next message from the mailslot.	
When finished with the mailslot, the creating process calls *DosDeleteMailslot*( ) to delete the mailslot and prevent future reception of data from senders.	

2.4 Named Pipe Service Model

2.4.1 Distributed Computing Model

Named pipes provide a bidirectional, connection-oriented interprocess communication mechanism. A named pipe is an object which appears to a client process in the file name space maintained by an LMX server. On the CAE system where the LMX implementation is running, this object is really a bidirectional communication channel that has been created by a CAE process to enable data exchange with a client process. Once the server application has created the named pipe, the client can open it via the name of the named pipe. After the connection is established either process may read or write using the named pipe handle associated with the opened named pipe, each reading what the other has written. To enable more than one client process to communicate with the server application the named pipe can be instanced. Using instance handles for a named pipe, multiple clients and a single server application can communicate without conflict. Each named pipe handle refers to a named pipe instance, all identified by the same name. Note that the client and server application are logical distinctions. In a multi-tasking environment, the client and server application may reside on the same system.

Access to a named pipe is controlled via the ''IPC$'' resource. The LMX implementation must offer the ''IPC$'' resource name before a server application can create a named pipe. The system running the client process must connect to that ''IPC$'' resource before a client process can open a named pipe.

2.4.2 Naming

The name for a pipe has the following form:

local: **\PIPE\pipename**
remote: **\\servername\PIPE\pipename**

servername identifies the LMX server local to the process creating the named pipe.

pipename the name assigned to the named pipe by the creating server application. Names for named pipes are IPC names.

2.4.3 Security

Security on named pipes can be performed using either share-level access or user-level access. For more information on these types of security see Section 2.1 on page 3 or refer to the referenced document **Protocols for X/Open PC Interworking: SMB**.

Access Control Lists

Some LMX server implementations provide access control lists (ACLs) in user-level security. An ACL may be created for any IPC name. An ACL lists all users and groups of users with their respective access rights for each object.

Because a named pipe appears in the file name space maintained by an LMX server, it is subject to the access validation of the server security system. The permissions defined for a named pipe on the server determine which users can open the pipe.

Other Mechanisms

LMX servers that do not implement ACLs may provide additional security via other mechanisms. A special user identification (ID) and group ID, *lmpipe* for example, could be defined by a server implementation for named pipe security. Only clients with the appropriate user ID or group ID would be allowed access to the named pipe.

In a CAE, it is desirable to limit which processes can create named pipes. This may be done by limiting named pipe creation to processes belonging to a special user ID or group ID, such as *lmpipe*, or to a process with appropriate privileges.

2.4.4 Pipe Instances and Inheritance

Multiple instances of a named pipe can be created by a server application. The first time *DosMakeNmPipe*( ) is called, the named pipe is created and a handle referencing that named pipe is returned. Each subsequent time that *DosMakeNmPipe*( ) is called for the same named pipe name (on the same LMX server), a new instance of the named pipe is created and a new handle referencing that instance is returned. The first call to *DosMakeNmPipe*( ) controls the number of instances that can be created for the named pipe.

Open named pipes and named pipe instances on the server may be inherited by child processes via *fork*( ) (see the **X/Open Portability Guide**, **Issue 3**, **Volume 2**, **XSI System Interface and Headers**) and are inherited via *LmForkNmPipe*( ). Details of named pipe inheritance are implementation-defined.

There is no process ownership of named pipe handles. Once a named pipe handle is created, either the original handle or a new instance handle, any process that inherits the handle can affect the state of that specific handle by issuing calls to *DosConnectNmPipe*( ) and *DosDisconnectNmPipe*( ).

2.4.5 Process Termination

All named pipes currently opened by a process are closed if a server application exits without calling *DosClose*( ). The time needed to clean up resources is implementation-dependent.

When the last CAE server application that is holding the original handle for the named pipe terminates, either via a normal exit condition or a process failure, the LMX server removes the name of the named pipe from the named pipe name space.

2.4.6 Modes

A named pipe is created as either a byte-mode or a message-mode pipe. A byte-mode pipe treats the named pipe like a simple stream of bytes; no boundaries between data sent in separate calls to *DosWrite*( ) are preserved. A message-mode pipe, however, does preserve those boundaries; all data in a given message is delivered as a unit.

For named pipes in byte stream mode only the API limits the amount of data that can be sent or received to 65535 bytes. For named pipes in message mode a single message cannot be larger than 65535 bytes. There is no limit on the amount of data or messages that can be sent with subsequent API calls.

A named pipe is created as inbound, outbound or bidirectional. An inbound named pipe transfers data from client to server. An outbound named pipe transfers data from server to client.

2.4.7 API Summary

The following table summarises the APIs for named pipe usage. The first column lists the API; the second column shows whether it may be issued by a client process, server process or both; and the third column is a short description. The client-only APIs (*DosOpen*( ), *DosWaitNmPipe*( ) and *DosCallNmPipe*( )) are optional on CAE systems.

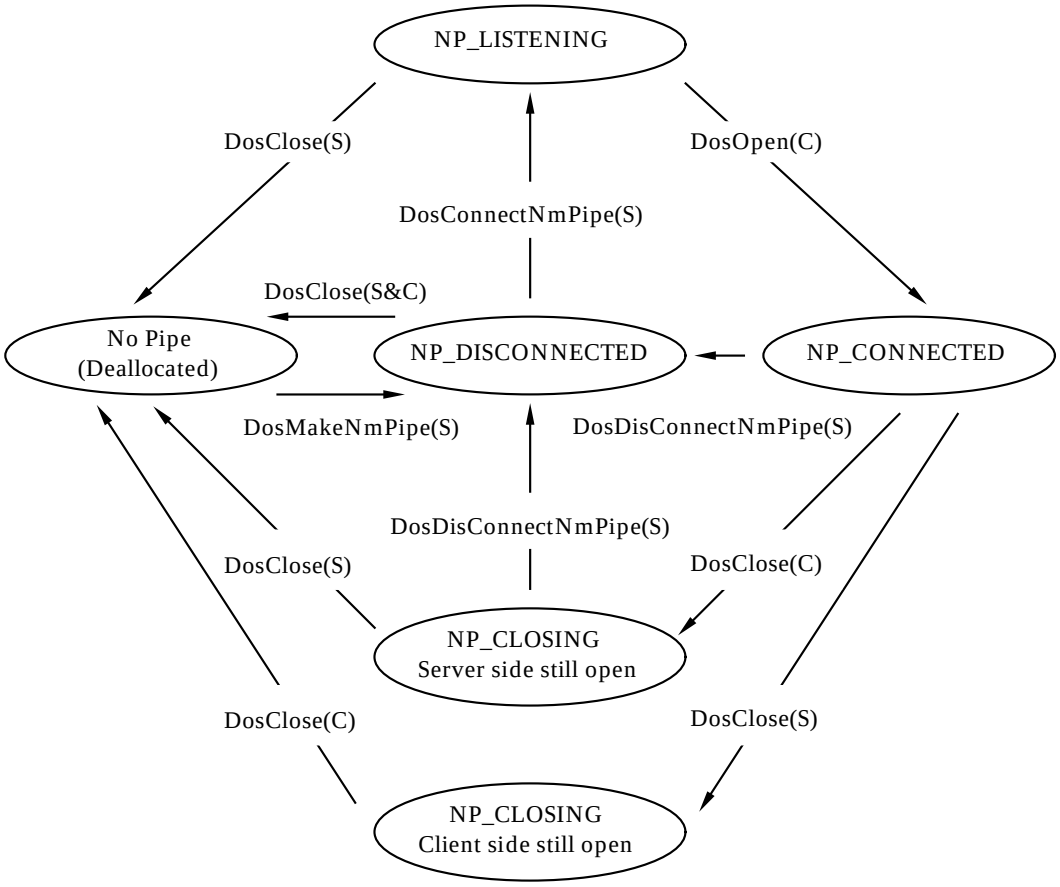
API	Issued by	Description
Creation:		
DosConnectNmPipe	server	Makes it possible for a client to open an instance of a named pipe.
DosDupHandle	both	Create a duplicate handle from an existing named pipe instance handle.
DosMakeNmPipe	server	Create a named pipe, or a new instance of an existing named pipe, and return its handle.
DosOpen	client	Open the client end of a named pipe and return its handle.
Management:		
DosQFHandState	both	Retrieve information about a named pipe handle.
DosQNmpHandState	both	Retrieve the current state of a named pipe handle.
DosQNmPipeInfo	both	Retrieve information about a named pipe.
DosSetFHandState	both	Modify certain open modes for a named pipe.
DosSetNmpHandState	both	Modify the state of a named pipe handle.
DosWaitNmPipe	client	Wait for the named pipe to become available.
LmForkNmPipe	server	Fork a child process, permitting the child to inherit the open handles.
Usage:		
DosBufReset	both	Flush a named pipe instance.
DosCallNmPipe	client	Open the client end of a named pipe, write data to the pipe, read data from the pipe, and close it.
DosPeekNmPipe	both	Read data from a named pipe instance without removing it.
DosRead	both	Read data from a named pipe.
DosTransactNmPipe	both	Write data to and read data from a named pipe.
DosWrite	both	Write data to a named pipe.
Removal:		
DosClose	both	Close a named pipe instance.
DosDisconnectNmPipe	server	Disconnect a named pipe handle.

2.4.8 States

Once it has been created, a named pipe instance can be in one of four states, defined in <*nmpipe.h*> on page 20.

State	Meaning
NP_DISCONNECTED	The pipe has been created or disconnected but is not ready for client access.
NP_LISTENING	A *DosConnectNmPipe*( ) call has been made, but the client end has not yet executed a *DosOpen*( ).
NP_CONNECTED	The client end has executed a *DosOpen*( ).
NP_CLOSING	Either the client has executed a *DosClose*( ) and the server has not yet executed a *DosDisconnectNmPipe*( ), or the server has executed a *DosClose*( ) and the client has not yet executed a *DosClose*( ).

The following diagram describes named pipe transition states:



2.4.9 Sample Named Pipe Communication Scenario

Server side	Client side
Make the ‘‘IPC$’’ share available for use by a client.	
	Connect to the ‘‘IPC$’’ share.
An LMX server running with user-level security may deny access to the ‘‘IPC$’’ share based on an ACL, user or group. An LMX server running with share-level security may deny access to the ‘‘IPC$’’ share based on a password.	
A server process for the distributed application calls *DosMakeNmPipe*( ) to create a named pipe with the specified name and characteristics.	
The application server process calls *DosConnectNmPipe*( ) to make the pipe available for client processes and to return a pipe handle to the server process for future operations on the pipe.	
	A client process for the distributed application calls *DosOpen*( ) to open the named pipe. The file handle returned on *DosOpen*( ) is used for future operations on the pipe.
	The client can send messages to the server via *DosWrite*( ).
The server reads these messages via *DosRead*( ) and responds with *DosWrite*( ).	
	The client receives the information from the server via *DosRead*( ).
	To terminate the connection, the client process may close the pipe handle by calling *DosClose*( ).
The application server process may call *DosDisconnectNmPipe*( ) to terminate the pipe instance and prevent the client further access.	
To remove the named pipe from the named pipe name space, the server process must perform a *DosClose*( ) on all instances for the named pipe, or *exit*( ).	

2.5 Messaging Service Model

2.5.1 Basic Model of the Messaging Service

This specification defines one side of the messaging service model in detail. That is, the side which generates messages using the *NetMessageBufferSend*( ) interface. The messaging service model is implementation-specific and typically has a server side that receives the information.

Using the *NetMessageBufferSend*( ) interface, information, which is generally assumed to be text, can be sent to services which either log the information or perform a *PopUp* feature on the screen informing the user of actions performed on his behalf, for example, a print job is complete.

2.5.2 Naming

Unlike naming for mailslots and named pipes, messaging uses two names. The first is the messaging name where the message is directed and the second is the recipient name. The recipient name indicates the intended messaging service for the information. Wildcarding for this name is possible. Names used for messaging are restricted in length (see *NetMessageBufferSend*( ) on page 65) and are case-sensitive.

2.5.3 Security

There is no security on the messaging service. Any user is capable of generating messages and directing them to any messaging service within the network.

2.5.4 Sample Messaging Communication Scenario

Server side	Client side
Perform action to allow for receipt of messages. The server receives the message.	A process calls *NetMessageBufferSend*( ) to send information to the message name.

2.5.5 API Summary

The following table summarises the APIs for messaging:

API	Issued by	Description
NetMessageBufferSend	sender	Send a buffer of data to a message name.

Chapter 3

Application Programming Interfaces

3.1 Introduction

This chapter defines the headers and Application Programming Interfaces (APIs) used by a CAE programmer to utilise the LMX interprocess communication (IPC) facilities. A general overview describes the SMB IPC facilities followed by a description of each API, including parameters and return codes.

General Information about the LMX APIs

The LMX APIs specified by X/Open can be divided into three categories: named pipes, mailslots and messaging.

Headers

#include <lm/lmerror.h>
#include <lm/mailslot.h>
#include <lm/message.h>
#include <lm/nmpipe.h>

The various return values which can be returned by the LMX APIs are defined in <*lmerror.h*> on page 16, and are identified by name only in manual pages describing the individual function calls. The LMX APIs will return one of the return values as specified in this file. Only these symbolic names should be used in programs since the actual value of the error number is implementation-defined. In case of an LMX API failure the return value will indicate the error condition.

Types and constants used by named pipes, mailslots and messaging are defined in <*nmpipe.h*> on page 20, <*mailslot.h*> on page 18 and <*message.h*> on page 19, respectively.

3.2 Include Files

The following pages describe the include files.

NAME

<lmerror.h> — LMX error definitions file.

SYNOPSIS

```
#include <lm/lmerror.h>
```

DESCRIPTION

The **<lm/lmerror.h>** header defines the constants for all of the ERROR_ names used within this specification. These are:

[ERROR_ACCESS_DENIED]	The system does not allow access to the file specified.
[ERROR_INVALID_ACCESS]	The access code is invalid.
[ERROR_INVALID_DATA]	The data is invalid.
[ERROR_INVALID_FUNCTION]	Incorrect function.
[ERROR_INVALID_HANDLE]	Incorrect internal file identifier.
[ERROR_FILE_NOT_FOUND]	The system cannot find the file specified.
[ERROR_NOT_ENOUGH_MEMORY]	Insufficient storage to process this command.
[ERROR_OPEN_FAILED]	The *DosOpen*( ) failed.
[ERROR_PATH_NOT_FOUND]	The system cannot find the path specified.
[ERROR_TOO_MANY_OPEN_FILES]	The system cannot open the file.
[NERR_Success]	No errors encountered.
[NO_ERROR]	Success, no error.
[ERROR_BAD_NETPATH]	The network path cannot be located.
[ERROR_DEV_NOT_EXIST]	This device does not exist on the network.
[ERROR_DUP_NAME]	A duplicate name exists on the network.
[ERROR_NETWORK_BUSY]	The network is busy.
[ERROR_NO_MORE_FILES]	There are no more files.
[ERROR_NOT_SUPPORTED]	The network request is not supported.
[ERROR_REM_NOT_LIST]	This remote computer is not listening.
[ERROR_BAD_NET_RESP]	The network has responded incorrectly.
[ERROR_NETNAME_DELETED]	The network name was deleted.
[ERROR_NETWORK_ACCESS_DENIED]	Network access is denied.
[ERROR_UNEXP_NET_ERR]	An unexpected network error has occurred.
[ERROR_BAD_NET_NAME]	A broadcast message cannot be made unless the *class* argument is 2.
[ERROR_FILE_EXISTS]	The file exists.
[ERROR_INVALID_PARAMETER]	The parameter is incorrect.
[ERROR_OUT_OF_STRUCTURES]	No more structures available. Implementation-specific error encountered.
[ERROR_SHARING_PAUSED]	File sharing has been temporarily paused.

[ERROR_BAD_FORMAT]	Pipe is a byte stream pipe or is not bidirectional.
[ERROR_BROKEN_PIPE]	The opposite end of the pipe has broken the connection.
[ERROR_BUFFER_OVERFLOW]	Buffer passed to system call is too small to hold return data.
[ERROR_INTERRUPT]	Interrupted system call.
[ERROR_INVALID_TARGET_HANDLE]	Invalid *newhandle* was specified. This error is implementation-dependent.
[ERROR_NO_ITEMS]	There were no items to operate upon.
[ERROR_NOT_READY]	Read on pipe in progress during flush.
[ERROR_SEM_TIMEOUT]	Timeout occurred.
[ERROR_INVALID_LEVEL]	Unimplemented level for information retrieval or setting.
[ERROR_INVALID_NAME]	Illegal character or malformed file system name.
[ERROR_PATH_BUSY]	All instances in use.
[ERROR_BAD_PIPE]	This is a non-existent pipe or an invalid operation.
[ERROR_MORE_DATA]	Additional data is available.
[ERROR_NO_DATA]	No data available for a non-blocking read.
[ERROR_PIPE_BUSY]	The specified pipe is busy.
[ERROR_PIPE_NOT_CONNECTED]	The pipe was disconnected by the server.
[ERROR_VC_DISCONNECTED]	The session was cancelled.

SEE ALSO

All API specifications within this document.

NAME

<mailslot.h>

SYNOPSIS

```
#include <lm/mailslot.h>
```

DESCRIPTION

The **<lm/mailslot.h>** header defines the API definition for the mailslot function calls.

SEE ALSO

DosDeleteMailslot( )	Delete a mailslot.
DosMailslotInfo( )	Retrieve information about a mailslot.
DosMakeMailslot( )	Create a mailslot and return its handle.
DosPeekMailslot( )	Read a message without removing it.
DosReadMailslot( )	Read the next message from the mailslot.
DosWriteMailslot( )	Send a message to a mailslot.

NAME

<message.h>

SYNOPSIS

```
#include <lm/message.h>
```

DESCRIPTION

The **<lm/message.h>** header defines the API definition for the messaging function call.

SEE ALSO

NetMessageBufferSend( )	Send a message to one or more users or applications.

NAME

<nmpipe.h>

SYNOPSIS

```
#include <lm/nmpipe.h>
```

DESCRIPTION

The **<lm/nmpipe.h>** header defines the API definition for the named pipe function calls and the following defines:

Manifest	Value	Description
NP_AMODE	0x0007	Access mode bit mask.
NP_INHERITANCE	0x0080	Inheritance bit.
NP_NBLK	0x8000	Non-blocking read or write.
NP_SERVER	0x4000	Set if server end.
NP_WMESG	0x0400	Write messages.
NP_RMESG	0x0100	Read as message.
NP_ICOUNT	0x00FF	Instance count field.
NP_DISCONNECTED	1	Disconnected state indicator.
NP_LISTENING	2	Listening state indicator.
NP_CONNECTED	3	Connected state indicator.
NP_CLOSING	4	Closing state indicator.

SEE ALSO

DosBufReset( )	Flush a named pipe instance.
DosCallNmPipe( )	Open the client end of a named pipe, write data to the pipe, read data from the pipe, and close it.
DosClose( )	Close a named pipe instance.
DosConnectNmPipe( )	Wait for a client to open an instance of a named pipe.
DosDisconnectNmPipe( )	Disconnect a named pipe instance.
DosDupHandle( )	Create a duplicate handle from an existing named pipe instance handle.
DosMakeNmPipe( )	Create a named pipe, or new instance of existing named pipe, and return handle.
DosOpen( )	Open the client end of a named pipe and return its handle.
DosPeekNmPipe( )	Read data from a named pipe instance without removing it.
DosQFHandState( )	Retrieve information about a named pipe handle.
DosQNmPipeInfo( )	Retrieve information about a named pipe.
DosQNmpHandState( )	Retrieve the current state of named pipe handle instance.
DosRead( )	Read data from a named pipe instance.
DosSetFHandState( )	Modify certain open modes for a named pipe.
DosSetNmpHandState( )	Modify the handle state of a named pipe instance.
DosTransactNmPipe( )	Write/read data to/from a named pipe instance.
DosWaitNmPipe( )	Wait for a named pipe to become available.
DosWrite( )	Write data to a named pipe instance.
LmForkNmPipe( )	Fork a CAE process that inherits named pipe handles.

3.3 Mailslots

The mailslot API provides access to the mailslot messaging IPC mechanism provided by LMX. (Note that mailslots are completely separate from any interpersonal mail mechanisms defined in the X/Open Portability Guide.)

A mailslot is a one-way form of interprocess communication. When writing to a mailslot, the sender must specify a numeric priority in the range 0 to 9, where 9 represents the highest priority. The receiving system will deliver messages in an order which is based on the priority. The priority ordering and handling is implementation-defined. It will always be true that a message with priority 9 is delivered before a message with priority 0. A server may have many mailslots in existence at any given time.

There are two classes of mailslot:

Class 1 Uses guaranteed means to transmit messages of up to 65535 bytes in length. Messages to class 1 mailslots may not be broadcast.

Class 2 Delivery is not guaranteed. Maximum length of a message depends on the configuration of the receiver, but will never be less than 360 bytes. Messages may be broadcast to class 2 mailslots; that is, a given message can be sent to a particular mailslot name on all systems. Support for broadcast messages may be configuration-dependent.

Mailslots are not inheritable by child processes, as a mailslot handle is not a real handle in the same sense as file handles and named pipe handles.

The semantics of a message sent to a particular mailslot are determined solely by the process reading the message from that mailslot.

Mailslot API Definitions

The following pages give the Mailslot API definitions.

NAME

DosDeleteMailslot( ) — deletes a mailslot, discarding all messages, whether or not they have been read.

SYNOPSIS

```
#include <lm/mailslot.h>
#include <lm/lmerror.h>

unsigned
DosDeleteMailslot(handle)

unsigned handle;
```

DESCRIPTION

Generally, a mailslot is deleted as the last step in the program's execution. Only the application that created the mailslot can delete it.

The parameters are as follows:

handle Specifies which mailslot to delete.

RETURN VALUE

Manifest	**Meaning**
[NERR_Success]	No errors encountered.
[ERROR_INVALID_HANDLE]	Invalid handle specified.

SEE ALSO

DosMakeMailslot( )	Create a mailslot and return its handle.
DosMailslotInfo( )	Retrieve information about a mailslot.

NAME

DosMailslotInfo( ) — retrieves information about a mailslot.

SYNOPSIS

```
#include <lm/mailslot.h>
#include <lm/lmerror.h>

unsigned
DosMailslotInfo(
                handle,
                messagesize,
                mailslotsize,
                nextsize,
                nextpriority,
                msgcount
               )

unsigned handle;
unsigned short * messagesize;
unsigned short * mailslotsize;
unsigned short * nextsize;
unsigned short * nextpriority;
unsigned short * msgcount;
```

DESCRIPTION

The parameters are as follows:

handle Specifies which mailslot to return information about.

messagesize Points to an unsigned short integer where the maximum size (in bytes) of message that the mailslot can accept is returned.

mailslotsize Points to an unsigned short integer where the size (in bytes) of the mailslot is returned. The parameter *mailslotsize* is equal to or exceeds *messagesize*.

nextsize Points to an unsigned short integer where the size (in bytes) of the next message in the mailslot is returned. If 0, no message is available.

nextpriority Points to an unsigned short integer where the priority of the next message in the mailslot is returned. Reference Section 3.3 on page 21 for more information on priorities. If *nextsize* is zero the value is undefined.

msgcount Points to an unsigned short integer where the count of messages the mailslot contains is returned.

RETURN VALUE

Manifest	**Meaning**
[NERR_Success]	No errors encountered.
[ERROR_INVALID_HANDLE]	Invalid handle specified.

SEE ALSO

DosMakeMailslot( )	Create a mailslot and return its handle.
DosWriteMailslot( )	Send a message to a mailslot.
DosReadMailslot( )	Read the next message from the mailslot.

NAME

DosMakeMailslot( ) — creates a mailslot and returns its handle.

SYNOPSIS

```
#include <lm/mailslot.h>
#include <lm/lmerror.h>

unsigned
DosMakeMailslot(
                name,
                messagesize,
                mailslotsize,
                handle
               )

char * name;
unsigned short messagesize;
unsigned short mailslotsize;
unsigned * handle;
```

DESCRIPTION

No two mailslots on any one system can have the same name. Mailslot handles are not inherited over either *fork*( ) or *LmForkNmPipe*( ).

The parameters are as follows:

name Points to an ASCIIZ string that assigns a name to the mailslot. The name must be of the form **\MAILSLOT\name**. The **name** is an IPC name (see Section 2.2 on page 4). The error code [ERROR_PATH_NOT_FOUND] is returned when the mailslot name is improperly formed.

messagesize Specifies the maximum message size (in bytes) that the mailslot can accept. Generally, mailslots cannot accept messages larger than some implementation-dependent value, but this limit is no smaller than 360 bytes. If the size specified is zero or greater than the maximum allowable size, the error [ERROR_INVALID_PARAMETER] is returned.

mailslotsize Specifies the size (in bytes) of the mailslot. The *mailslotsize* parameter must equal or exceed *messagesize*, otherwise [ERROR_INVALID_PARAMETER] is returned. This parameter is advisory to the system for preallocation of resources.

handle Points to a unsigned integer which is the returned handle for the mailslot.

RETURN VALUE

Manifest	**Meaning**
[NERR_Success]	No errors encountered.
[ERROR_FILE_EXISTS]	The mailslot already exists.
[ERROR_INVALID_PARAMETER]	Invalid parameter specified.
[ERROR_NOT_ENOUGH_MEMORY]	Not enough memory for operation.
[ERROR_PATH_NOT_FOUND]	Unknown pathname specified.

SEE ALSO

DosDeleteMailslot( )	Delete a mailslot.
DosMailslotInfo( )	Retrieve information about a mailslot.

NOTES

The minimum upper bound of 360 bytes on *messagesize* is based upon a minimum guaranteed datagram size of 512 bytes minus protocol overhead. Any implementation based on the RFC 1001 and RFC 1002 (published in the referenced document **Protocols for X/Open PC Interworking: SMB**) has this limit.

NAME

DosPeekMailslot( ) — reads the most current message, based on priority, in a mailslot without removing it.

SYNOPSIS

```
#include <lm/mailslot.h>
#include <lm/lmerror.h>

unsigned
DosPeekMailslot(
                handle,
                buf,
                bytesread,
                nextsize,
                nextpriority
               )

unsigned handle;
char * buf;
unsigned short * bytesread;
unsigned short * nextsize;
unsigned short * nextpriority;
```

DESCRIPTION

If a higher-priority message arrives, there is no guarantee that a message previously read by the *DosPeekMailslot*( ) function is the same message read by a subsequent call to the *DosReadMailslot*( ) function. See Section 3.3 on page 21 for more information on priorities.

The parameters are as follows:

handle Specifies which mailslot is to be read.

buf Points to the buffer for the returned message. The space pointed to by *buf* must be at least as large as the *messagesize* parameter passed to the *DosMakeMailslot*( ) function that created the mailslot.

bytesread Points to an unsigned short integer where the size (in bytes) of the returned message is returned. If no message is available, *bytesread* is 0.

nextsize Points to an unsigned short integer where the size (in bytes) of the next message in the mailslot is returned. If the mailslot contains no other message, *nextsize* is 0.

nextpriority Points to an unsigned short integer where the priority of the next message in the mailslot is returned. Refer to Section 3.3 on page 21 for more information on priorities. If *nextsize* is zero the value is undefined.

RETURN VALUE

Manifest	**Meaning**
[NERR_Success]	No errors encountered.
[ERROR_INVALID_HANDLE]	Invalid handle specified.
[ERROR_BROKEN_PIPE]	Data written to a Class 1 mailslot with no reader.

SEE ALSO

DosReadMailslot( )	Read the next message from the mailslot.
DosWriteMailslot( )	Send a message to a mailslot.
DosMakeMailslot( )	Create a mailslot and return its handle.

NAME

DosReadMailslot( ) — reads, then removes, a mailslot's message.

SYNOPSIS

```
#include <lm/mailslot.h>
#include <lm/lmerror.h>

unsigned
DosReadMailslot(
                handle,
                buf,
                bytesread,
                nextsize,
                nextpriority,
                timeout
               )

unsigned handle;
char * buf;
unsigned short * bytesread;
unsigned short * nextsize;
unsigned short * nextpriority;
long timeout;
```

DESCRIPTION

Messages are stored in a mailslot based on their priority. An incoming message with a higher priority is stored ahead of a previously stored message with the same or lower priority. *DosReadMailslot*( ) always reads and removes the highest priority message. Refer to Section 3.3 on page 21 for more information on priorities. It is possible to specify the amount of time the caller waits for the mailslot message. If this amount of time passes without receiving any mailslot messages, *DosReadMailslot*( ) returns with the error [ERROR_SEM_TIMEOUT].

The parameters are as follows:

handle Specifies the handle for the mailslot read operation.

buf Points to the buffer for the returned message. The space pointed to must contain at least as many bytes as specified in the *messagesize* parameter passed to the *DosMakeMailslot*( ) function.

bytesread Points to an unsigned short integer where the size (in bytes) of the returned message is returned. If 0, no message is available.

nextsize Points to an unsigned short integer where the size (in bytes) of the next message in the mailslot is returned. If 0, the mailslot contains no more messages.

nextpriority Points to an unsigned short integer where the priority of the next message is returned (undefined if *nextsize* is 0). Reference Section 3.3 on page 21 for more information on priorities.

timeout A long integer indicating to the function how many milliseconds to wait if a message is not available immediately. If 0, *DosReadMailslot*( ) returns immediately if no message is present. If –1, *DosReadMailslot*( ) waits indefinitely.

RETURN VALUE

Manifest	**Meaning**
[NERR_Success]	No errors encountered.

[ERROR_INVALID_HANDLE]	Invalid handle specified.
[ERROR_SEM_TIMEOUT]	Timeout occurred.
[ERROR_BROKEN_PIPE]	Data written to a Class 1 mailslot with no reader.

SEE ALSO

DosPeekMailslot( )	Read a message without removing it.
DosWriteMailslot( )	Send a message to a mailslot.
DosMakeMailslot( )	Create a mailslot and return its handle.

NAME

DosWriteMailslot( ) — writes a message to a particular mailslot.

SYNOPSIS

```
#include <lm/mailslot.h>
#include <lm/lmerror.h>

unsigned
DosWriteMailslot(
                 name,
                 message,
                 size,
                 priority,
                 class,
                 timeout
                )

char * name;
char * message;
unsigned short size;
unsigned short priority;
unsigned short class;
long timeout;
```

DESCRIPTION

DosWriteMailslot( ) fails if there is not enough room to write *message* in the mailslot.

Messages may be subject to implementation-dependent length limitations when writing to remote mailslots. Messages of at least 360 bytes are supported. Messages can be any size up to 65535 bytes when written to local mailslots.

When writing to a mailslot, the sender must specify a numeric priority. The receiving system delivers messages in an order which is based on the priority. Refer to Section 3.3 on page 21 for more information on priorities.

The parameters are as follows:

name Points to an ASCIIZ string containing the name of the mailslot to which the message is to be written. For a local mailslot, use the format **\MAILSLOT\name**; for a remote mailslot the format **\\computername\MAILSLOT\name** is used where **computername** follows the rules for a computer name and **name** follows the rules for an IPC name (see Section 2.2 on page 4). For all mailslots with a particular name, but on different computers, use the format ***\MAILSLOT\name**. If an attempt is made to send to all mailslots of a particular name on different computers using the Class 1 form of mailslot support the return value [ERROR_BAD_NET_NAME] is returned and the message is not sent.

message Points to a buffer containing the message to be written to the mailslot.

size Specifies the size (in bytes) of *message*. Zero length messages are not supported. If an attempt is made to send a zero length message the return value [ERROR_INVALID_PARAMETER] is returned.

priority Assigns a priority to the message. Refer to Section 3.3 on page 21 for more information on priorities.

class Specifies the class of mailslot service. Only Class 2 mailslots are supported by all systems. If Class 1 mailslots are not supported the return value [ERROR_INVALID_PARAMETER] is returned by this function.

timeout Specifies the number of milliseconds to attempt writing a message to a mailslot. If 0, *DosWriteMailslot*( ) attempts to write the message only once. If –1, *DosWriteMailslot*( ) attempts to write a message to a mailslot for an indefinite time.

RETURN VALUE

Manifest	**Meaning**
[NERR_Success]	No errors encountered.
[ERROR_BAD_NET_NAME]	A broadcast message cannot be made unless the *class* argument is 2.
[ERROR_BROKEN_PIPE]	Data written to a Class 1 mailslot with no reader.
[ERROR_BUFFER_OVERFLOW]	Buffer too small.
[ERROR_INVALID_PARAMETER]	Invalid parameter specified.
[ERROR_PATH_NOT_FOUND]	Unknown pathname specified.
[ERROR_SEM_TIMEOUT]	Timeout occurred.

SEE ALSO

DosMakeMailslot( )	Create a mailslot and return its handle.
DosReadMailslot( )	Read the next message from the mailslot.

NOTES

The length limitation of 360 for remote workstations is based on least-common denominator memory restrictions for existing client implementations. For remote servers, the length limitation is related to the protocol stack chosen and its restrictions on datagram length.

3.4 Named Pipes

The named pipes API provides access to the named pipes IPC mechanism provided by LMX. (Note that the named pipes referred to here are quite different and distinct from FIFOs as defined in the **X/Open Portability Guide**, **Issue 3**, **Volume 2**, **XSI System Interface and Headers**.

A named pipe is an object which appears to a PC client as a file in the set of files maintained by an LMX server. On the server, this object is a communications channel that has been created by a server application. The named pipe can be instanced, permitting several clients to communicate over a particular named pipe without conflict. Multiple instances of a named pipe can be created by a server application. The first time *DosMakeNmPipe*( ) is called, the named pipe will be created and a handle referencing that named pipe is returned. Each subsequent time that *DosMakeNmPipe*( ) is called for the same named pipe name (on the same LMX server), a new instance of the named pipe is created and a new handle referencing that instance is returned. The first call to *DosMakeNmPipe*( ) controls the number of instances that can be created for the named pipe.

Pipe handles are inheritable by child processes. It is possible for an LMX implementation to support the inheritance of created named pipes across a *fork*( ) function call. When *LmForkNmPipe*( ) is used all named pipe instances are guaranteed to be inherited by the child process.

A named pipe is either a byte-stream-mode or message-mode pipe. A byte-stream-mode pipe treats the named pipe like a simple stream of bytes; no boundaries between data sent in separate calls to *DosWrite*( ) are preserved. A message-mode pipe, however, does preserve those boundaries; all data in a given message will be delivered as a unit. It is possible for the reader of a message-mode pipe to interpret the data as a stream instead.

3.4.1 Named Pipe States

Once it has been created, a named pipe instance can be in one of four states, defined in <*nmpipe.h*> on page 20:

State	Event Causing State Entry
NP_DISCONNECTED	Server invoked *DosCreateNmPipe*( ) or *DosDisconnectNmPipe*( ).
NP_LISTENING	Server invoked *DosNmPipeConnect*( ).
NP_CONNECTED	Client process opened pipe and was connected to this instance.
NP_CLOSING	Client or server process closed its end of the pipe.

Named Pipe API Definitions

The following pages contain the Named Pipe API definitions.

NAME

DosBufReset( ) — called to flush data on a named pipe instance.

SYNOPSIS

```
#include <lm/lmerror.h>
#include <lm/nmpipe.h>

unsigned
DosBufReset(handle)

unsigned handle;
```

DESCRIPTION

If a client process is reading data from the named pipe instance when *DosBufReset*( ) is called, the function blocks the calling process until all of the data is read. If called on the server side, *DosBufReset*( ) times out and returns [ERROR_NOT_READY] if the data is not read within an implementation-defined interval.

The parameters are as follows:

handle Specifies which named pipe instance to flush.

RETURN VALUES

Manifest	**Meaning**
[NERR_Success]	No errors encountered.
[ERROR_INVALID_HANDLE]	Invalid handle specified.
[ERROR_NOT_READY]	Read on pipe in progress during flush.
[ERROR_PIPE_NOT_CONNECTED]	Named pipe is currently not in the NP_CONNECTED state.

SEE ALSO

DosClose( )	Close a named pipe instance.
DosWrite( )	Write data to a named pipe instance.
DosMakeNmPipe( )	Create a named pipe, or new instance of existing named pipe, and return handle.

NAME

DosCallNmPipe( ) — called by a client process to open a named pipe, write data to the pipe, read data from the pipe, and close it.

SYNOPSIS

```
#include <lm/lmerror.h>
#include <lm/nmpipe.h>

unsigned
DosCallNmPipe(
              name,
              writebuf,
              writelen,
              readbuf,
              readlen,
              bytesread,
              timeout
             )

char *name;
char *writebuf;
unsigned short writelen;
char *readbuf
unsigned short readlen;
unsigned short *bytesread;
long timeout;
```

DESCRIPTION

For this function to succeed the named pipe must be a bidirectional message mode pipe. If this is not true [ERROR_BAD_FORMAT] is returned. If the amount of data to be returned is greater than *readlen* then the data that fits into the *readbuf* area is returned, [ERROR_MORE_DATA] is returned and all other data is lost.

The parameters are as follows:

name Points to an ASCIIZ string containing the name of a pipe in the format **\PIPE\name** for a local pipe or **\\LMXservername\PIPE\name** for a remote pipe. Refer to the definition for IPC resource names in Section 2.2 on page 4.

writebuf Points to a buffer containing the data to be written.

writelen Specifies the size of the *writebuf* buffer in bytes.

readbuf Points to a buffer where the data returned is to be placed.

readlen Specifies the size of the *readbuf* buffer in bytes.

bytesread Indicates how many bytes of data were returned in *readbuf*.

timeout Specifies how many milliseconds to wait for the named pipe to become available. If the value is –1, the default time (set by *DosMakeNmPipe*( )) is used.

RETURN VALUE

Manifest	Meaning
[NERR_Success]	No errors encountered.
[ERROR_ACCESS_DENIED]	Access not allowed.
[ERROR_BAD_FORMAT]	Pipe is a byte stream pipe or is not bidirectional.
[ERROR_BROKEN_PIPE]	The pipe has been closed or has no reader.
[ERROR_INTERRUPT]	System call interrupted.
[ERROR_INVALID_NAME]	Syntax of name provided was invalid.
[ERROR_INVALID_PARAMETER]	Invalid parameter specified.
[ERROR_MORE_DATA]	More data available, buffer too small.
[ERROR_NOT_ENOUGH_MEMORY]	Not enough memory for operation.
[ERROR_OUT_OF_STRUCTURES]	No more structures available. Implementation-specific error encountered.
[ERROR_PATH_NOT_FOUND]	Unknown pathname specified.
[ERROR_PIPE_BUSY]	The specified pipe is busy.
[ERROR_PIPE_NOT_CONNECTED]	Named pipe is currently not in the NP_CONNECTED state.
[ERROR_SEM_TIMEOUT]	Timeout occurred.

SEE ALSO

DosClose( )	Close a named pipe instance.
DosMakeNmPipe( )	Create a named pipe, or new instance of existing named pipe, and return handle.
DosOpen( )	Open the client end of a named pipe and return its handle.
DosTransactNmPipe( )	Write/read data to/from a named pipe instance.

NAME

DosClose( ) — closes a named pipe instance.

SYNOPSIS

```
#include <lm/lmerror.h>
#include <lm/nmpipe.h>

unsigned
DosClose(handle)

unsigned handle;
```

DESCRIPTION

When the *DosClose*( ) function is called on either the client or server side of a named pipe, the named pipe instance is closed and the other end of the named pipe is notified of this via the error code [ERROR_BROKEN_PIPE] when another attempt is made to read or write to the named pipe. After the call to *DosClose*( ) the instance handle is invalid. If the *DosClose*( ) call is on the server side and this is the last instance for the named pipe, the named pipe's name is freed and no longer exists.

If a process exits an implicit *DosClose*( ) occurs for all open named pipe instances.

The parameters are as follows:

handle Specifies the handle of the named pipe instance to close.

RETURN VALUE

Manifest	**Meaning**
[NERR_Success]	No errors encountered.
[ERROR_BAD_PIPE]	Handle does not refer to a named pipe instance.
[ERROR_BROKEN_PIPE]	The pipe has been closed or has no reader.
[ERROR_INVALID_HANDLE]	Invalid handle specified.

SEE ALSO

DosDisconnectNmPipe( )	Disconnect a named pipe instance.

NAME

DosConnectNmPipe( ) — waits for a client process to open, via *DosOpen*( ), an instance of a named pipe.

SYNOPSIS

```
#include <lm/lmerror.h>
#include <lm/nmpipe.h>

unsigned
DosConnectNmPipe(handle)

unsigned handle;
```

DESCRIPTION

DosConnectNmPipe( ) may only be executed by the server application that created the named pipe, otherwise [ERROR_INVALID_FUNCTION] is returned.

If the instance is already connected the *DosConnectNmPipe*( ) has no effect and immediately returns [NERR_Success].

If the instance is not opened by the client and the pipe is in blocking mode, then *DosConnectNmPipe*( ) blocks until a client process opens the named pipe. The state of the named pipe instance is changed to NP_CONNECTED.

If the instance is not opened by the client and the named pipe is in non-blocking mode, then *DosConnectNmPipe*( ) places the instance in the NP_LISTENING state (see Section 2.4.8 on page 10) and immediately returns with [ERROR_PIPE_NOT_CONNECTED]. Subsequent *DosConnectNmPipe*( ) calls can be made to poll the named pipe instance. [ERROR_PIPE_NOT_CONNECTED] is returned until a client opens the named pipe and the named pipe instance's state changes to NP_CONNECTED. A subsequent *DosConnectNmPipe*( ) returns [NERR_Success].

If a named pipe instance that was previously opened by a client process has been closed but not disconnected by the server process, *DosConnectNmPipe*( ) returns the [ERROR_BROKEN_PIPE] error code. The connection must be removed via *DosDisconnectNmPipe*( ) before a new connection can be made.

The *blocking* and *non-blocking* mode is set by the *omode* of *DosMakeNmPipe*( ) and can be adjusted by calling *DosSetNmpHandState*( ).

The parameters are as follows:

handle Specifies the handle for the named pipe instance, as returned by the *DosMakeNmPipe*( ) function.

RETURN VALUE

Manifest	**Meaning**
[NERR_Success]	No errors encountered.
[ERROR_BAD_PIPE]	Handle does not refer to a named pipe instance.
[ERROR_BROKEN_PIPE]	The pipe has been closed or has no reader.
[ERROR_INTERRUPT]	System call interrupted.
[ERROR_INVALID_FUNCTION]	Attempt to invoke on client end of pipe.
[ERROR_INVALID_HANDLE]	Invalid handle specified.
[ERROR_PIPE_NOT_CONNECTED]	Named pipe is currently not in the NP_CONNECTED state.

SEE ALSO

DosDisconnectNmPipe( )	Disconnect a named pipe instance.
DosMakeNmPipe( )	Create a named pipe, or new instance of existing named pipe, and return handle.
DosSetNmpHandState( )	Modify the handle state of a named pipe instance.

NAME

DosDisconnectNmPipe( ) — called by a server application to disconnect a named pipe instance, denying a client process any further access to it.

SYNOPSIS

```
#include <lm/lmerror.h>
#include <lm/nmpipe.h>

unsigned
DosDisconnectNmPipe(handle)

unsigned handle;
```

DESCRIPTION

After the server call, the named pipe is placed in the NP_DISCONNECTED state.

DosDisconnectNmPipe( ) can only be executed by the server application which created the named pipe. When called by any other process, the function returns the [ERROR_INVALID_FUNCTION] error code.

If a client process has the named pipe instance open when *DosDisconnectNmPipe*( ) is called, the client handle remains open and is placed in the NP_DISCONNECTED state.

DosDisconnectNmPipe( ) discards all unread queued data from a named pipe instance.

RETURN VALUE

Manifest	**Meaning**
[NERR_Success]	No errors encountered.
[ERROR_BAD_PIPE]	Handle does not refer to a named pipe instance.
[ERROR_INVALID_FUNCTION]	Attempt to invoke on client end of pipe.
[ERROR_INVALID_HANDLE]	Invalid handle specified.

SEE ALSO

DosClose( )	Close a named pipe instance.
DosConnectNmPipe( )	Wait for a client to open an instance of a named pipe.

NAME

DosDupHandle( ) — creates a duplicate of a handle to a named pipe instance.

SYNOPSIS

```
#include <lm/lmerror.h>
#include <lm/nmpipe.h>

unsigned
DosDupHandle(
             oldhandle,
             newhandle
            )

unsigned   oldhandle;
unsigned * newhandle;
```

DESCRIPTION

Duplicating a handle duplicates all handle-specific information between *oldhandle* and *newhandle*. Thus, the original handle and the duplicate handle are interchangeable; changes to one handle affect the other. If the parameter *newhandle* specifies a valid named pipe handle, *DosDupHandle*( ) closes this handle via *DosClose*( ) prior to performing the duplication operation. If *newhandle* specifies a CAE file handle the action taken by this function is implementation-defined.

The parameters are as follows:

oldhandle Specifies the original handle of a named pipe instance.

newhandle Points to an unsigned integer specifying the new handle for the named pipe instance.

RETURN VALUE

Manifest	**Meaning**
[NERR_Success]	No errors encountered.
[ERROR_BAD_PIPE]	Handle does not refer to a named pipe instance.
[ERROR_INVALID_HANDLE]	Invalid handle specified.
[ERROR_INVALID_TARGET_HANDLE]	Invalid *newhandle* was specified. This error is implementation-dependent.

SEE ALSO

DosMakeNmPipe( )	Create a named pipe, or new instance of existing named pipe, and return handle.

NAME

DosMakeNmPipe( ) — creates a new named pipe or a new instance of an existing named pipe, and returns its handle.

SYNOPSIS

```
#include <lm/lmerror.h>
#include <lm/nmpipe.h>

unsigned
DosMakeNmPipe(
              name,
              handle,
              omode,
              pmode,
              outsize,
              insize,
              timeout
             )

char * name;
unsigned * handle;
unsigned short omode;
unsigned short pmode;
unsigned short outsize;
unsigned short insize;
long timeout;
```

DESCRIPTION

If multiple instances of a named pipe are created and they are in an NP_LISTENING state (via *DosConnectNmPipe*( )), several clients can simultaneously open instances and receive a handle to them by calling the *DosOpen*( ) function.

The parameters are as follows:

name Points to an ASCIIZ string containing the name of a local pipe in the format **\PIPE\name**. Legal names are IPC resource names (see Section 2.3.2 on page 6).

handle Specifies the handle for the new named pipe or new instance of a previously created named pipe.

omode A bitmap that specifies the open modes for the named pipe, defined as follows:

Manifest	Bit Mask	Meaning
NP_AMODE	0x0007	Sets access mode. If 0, pipe is inbound, going from client to server. If 1, pipe is outbound, going from server to client. If 2, pipe is bidirectional.
	0x0078	Reserved, with the value 0.

NP_INHERITANCE	0x0080	Sets inheritance mode. (See *LmForkNmPipe*( ) on page 63.) If 0, spawned processes inherit the named pipe's handle. If 1, spawned processes cannot inherit the handle.
	0x3F00	Reserved, with the value 0.
NP_WB	0x4000	Sets write behaviour. If 0, write-behind to remote named pipes is allowed. If 1, write-behind to remote named pipes is not allowed. Behaviour for this mode is implementation-defined.
	0x8000	Reserved, with the value 0.

pmode A bitmap specifying the pipe open mode for the named pipe, defined in <*nmpipe.h*> on page 20 as follows:

Manifest	**Bit Mask**	**Meaning**
NP_ICOUNT	0x00FF	Sets instance count (maximum number of concurrent openings allowed on this named pipe name). If 0xFF, then an implementation-defined number of instances can be created. The value 0 is reserved. The instance count is only required when the named pipe is created; in subsequent *DosMakeNmPipe*( ) calls for the same pipe this field is ignored.
NP_RMESG	0x0100	Sets pipe read mode. If 0, the pipe can be read as a byte-stream pipe. If 1, the pipe can be read either as a byte-stream or a message-mode pipe. This mode can be changed by calling the *DosSetNmpHandState*( ) function.
NP_WMESG	0x0400	Sets pipe type. If 0, the pipe is a byte-stream pipe. If 1, the pipe is a message-stream pipe.
NP_SERVER	0x4000	Must be 1. This bit is ignored as input to the *DosMakeNmPipe*( ) call.

NP_NBLK	0x8000	Sets blocking mode. If 0 (blocking mode), all reads and writes block when no data is available. Reads from a named pipe block until at least some data is available. Writes to a named pipe block until all data is written. If 1 (non-blocking mode), all reads and writes return immediately when no data is available. The blocking mode can be reset by calling the *DosSetNmpHandState*( ) function.
-		All other bits reserved, values undefined.

outsize Specifies a recommendation on how many bytes to allocate for the named pipe's outgoing buffer.

insize Specifies a recommendation on how many bytes to allocate for the named pipe's incoming buffer.

timeout Specifies the default timeout parameter (in milliseconds) for the *DosWaitNmPipe*( ) function to be used by the client. If 0, a default value is used. *timeout* takes effect on the first invocation of *DosWaitNmPipe*( ). For all subsequent invocations with the same named pipe name this parameter is ignored.

RETURN VALUE

Manifest	**Meaning**
[NERR_Success]	No errors encountered.
[ERROR_INVALID_NAME]	Syntax of name provided was invalid.
[ERROR_INVALID_PARAMETER]	Invalid parameter specified.
[ERROR_NOT_ENOUGH_MEMORY]	Not enough memory for operation.
[ERROR_OUT_OF_STRUCTURES]	No more structures available. Implementation-specific error encountered.
[ERROR_PATH_NOT_FOUND]	Unknown pathname specified.
[ERROR_PIPE_BUSY]	All instances in use.

SEE ALSO

DosConnectNmPipe( )	Wait for a client to open an instance of a named pipe.
DosOpen( )	Open the client end of a named pipe and return its handle.
DosQNmpHandState( )	Retrieve the current state of named pipe handle instance.
DosSetNmpHandState( )	Modify the handle state of a named pipe instance.
DosWaitNmPipe( )	Wait for a named pipe to become available.
LmForkNmPipe( )	Fork a CAE process that inherits named pipe handles.

NAME

DosOpen( ) — called by a client process to open the client process end of a named pipe and return its handle.

SYNOPSIS

```
#include <lm/nmpipe.h>
#include <lm/lmerror.h>

unsigned
DosOpen(
        name,
        handle,
        action,
        size,
        attribute,
        openflag,
        omode,
        reserved
       )

char * name;
unsigned * handle;
unsigned short * action;
unsigned long size;
unsigned attribute;
unsigned short openflag;
unsigned short omode;
unsigned long reserved;
```

DESCRIPTION

If the named pipe being opened was created as an *outbound* pipe and an attempt is made to open the pipe for writing, the error [ERROR_ACCESS_DENIED] is returned. If the named pipe being opened was created as an *inbound* pipe and an attempt is made to open the pipe for reading, the error [ERROR_ACCESS_DENIED] is returned. The mode of the named pipe after *DosOpen*( ) is what was specified via *DosMakeNmPipe*( ).

The parameters are as follows:

name Points to an ASCIIZ string containing the name of a pipe in the format **\PIPE\name** for a local pipe or **\\LMXservername\PIPE\name** for a remote pipe.

handle Points to the unsigned integer where the handle for the new named pipe instance is returned.

action Points to the unsigned short in which *DosOpen*( ) returns the action taken in the open. A value of 1 indicates that the named pipe was opened; any other value indicates that the open failed.

size Ignored, should be zero.

attribute Specifies the attributes of the named pipe. A value of 0 indicates read and write. A value of 1 indicates read only. There is no value to indicate write only.

openflag Specifies the action to take if the named pipe exists. A value of 0 indicates that the *DosOpen*( ) call should fail with return value [ERROR_OPEN_FAILED] if the specified name exists. A value of 1 indicates that the named pipe is to be opened if it exists.

omode A bitmap that specifies the open modes for the named pipe, defined as follows:

Manifest	Bit Mask	Meaning
NP_AMODE	0x0007	Sets access mode. If 0, pipe is inbound, going from client to server. If 1, pipe is outbound, going from server to client. If 2, pipe is full duplex, going both to and from server and client.
NP_INHERITANCE	0x0080	Sets inheritance mode. (See *LmForkNmPipe*( ) on page 63.) If 0, spawned processes inherit the named pipe's handle. If 1, spawned processes cannot inherit the handle.
NP_WB	0x4000	Sets write behaviour. If 0, write-behind to remote pipes is allowed. If 1, write-behind to remote pipes is not allowed.
-		All other bits reserved, values undefined.

reserved MBZ.

RETURN VALUE

Manifest	Meaning
[NERR_Success]	No errors encountered.
[ERROR_ACCESS_DENIED]	Access not allowed.
[ERROR_NOT_ENOUGH_MEMORY]	Not enough memory for operation.
[ERROR_OPEN_FAILED]	Open failed due to the existence of named pipe name and *openflag* set to 0.
[ERROR_PATH_NOT_FOUND]	Unknown pathname specified.
[ERROR_PIPE_BUSY]	All instances in use.

SEE ALSO

DosClose( )	Close a named pipe instance.
DosRead( )	Read data from a named pipe instance.
DosSetNmpHandState( )	Modify the handle state of a named pipe instance.
DosWrite( )	Write data to a named pipe instance.
LmForkNmPipe( )	Fork a CAE process that inherits named pipe handles.

NAME

DosPeekNmPipe( ) — performs a non-destructive read of the data from a named pipe instance.

SYNOPSIS

```
#include <lm/nmpipe.h>
#include <lm/lmerror.h>

unsigned
DosPeekNmPipe(
              handle,
              buf,
              buflen,
              bytesread,
              bytesavail,
              status
             )

unsigned handle;
char *buf;
unsigned short buflen;
unsigned short *bytesread;
unsigned short  bytesavail[2];
unsigned short *status;
```

DESCRIPTION

DosPeekNmPipe( ) acts like *DosRead*( ) with the following exceptions:

- The bytes read are not removed from the named pipe.
- *DosPeekNmPipe*( ) may return for a named pipe in message mode only part of the message (that part currently in the named pipe), even if the size of the peek would accommodate the whole message.
- *DosPeekNmPipe*( ) never blocks regardless of the blocking mode.
- Additional information about the status of the named pipe and remaining data is returned. This can be used to determine whether all of the current message has been returned. The named pipe is at "End-of-File" if no bytes are returned and *status* is NP_CLOSING or NP_DISCONNECTED.

The parameters are as follows:

handle	Specifies the handle of the named pipe instance.
buf	Points to the buffer where data is to be returned.
buflen	Specifies the size (in bytes) of the buffer.
bytesread	Points to an unsigned short integer telling how many bytes were read.
bytesavail	Points to an array of two unsigned shorts. The function returns the number of bytes left in the named pipe in bytesavail[0] and the number of bytes left in the current message in bytesavail[1].
status	Points to an unsigned short where the state of the named pipe is returned. Refer to Section 3.4.1 on page 33.

If *DosPeekNmPipe*( ) returns successfully with *bytesread* set to 0 then no data is available.

RETURN VALUE

Manifest	Meaning
[NERR_Success]	No errors encountered.
[ERROR_ACCESS_DENIED]	Access not allowed.
[ERROR_BAD_PIPE]	Handle does not refer to a named pipe instance.
[ERROR_BROKEN_PIPE]	The pipe has been closed or has no reader.
[ERROR_INVALID_HANDLE]	Invalid handle specified.
[ERROR_MORE_DATA]	More data available, buffer too small.
[ERROR_PIPE_BUSY]	The specified pipe is busy.
[ERROR_PIPE_NOT_CONNECTED]	Named pipe is currently not in the NP_CONNECTED state.

SEE ALSO

DosRead( )	Read data from a named pipe instance.

NAME

DosQFHandState( ) — retrieves information about the named pipe instance.

SYNOPSIS

```
#include <lm/nmpipe.h>
#include <lm/lmerror.h>

unsigned
DosQFHandState(
               handle,
               handlestate
              )

unsigned handle;
unsigned short * handlestate;
```

DESCRIPTION

The parameters are as follows:

handle Specifies the handle of the named pipe to be queried.

handlestate Points to an unsigned integer where the state of the specified pipe handle, defined as follows, is returned.

Manifest	Bit Mask	Meaning
NP_INHERITANCE	0x0080	If 0, spawned processes can inherit the named pipe's handle. If 1, spawned processes cannot inherit the handle.
NP_WB	0x4000	If 0, write-behind to a remote pipe is allowed. If 1, write-behind to a remote pipe is not allowed.
-		All other bits reserved, values undefined.

RETURN VALUE

Manifest	Meaning
[NERR_Success]	No errors encountered.
[ERROR_INVALID_HANDLE]	Invalid handle specified.

SEE ALSO

DosQNmpHandState( )	Retrieve the current state of named pipe handle instance.
LmForkNmPipe( )	Fork a CAE process that inherits named pipe handles.

NAME

DosQNmpHandState( ) — returns information about the current state of a named pipe instance handle.

SYNOPSIS

```
#include <lm/nmpipe.h>
#include <lm/lmerror.h>

unsigned
DosQNmpHandState(
                 handle,
                 pmode
                )

unsigned handle;
unsigned short *pmode;
```

DESCRIPTION

When invoked by the server process of a named pipe, the state bits reflect values established by the *DosMakeNmPipe*( ) function or subsequent calls to the *DosSetNmpHandState*( ) function. When invoked by the client process, the state bits reflect values established by the *DosOpen*( ) function or subsequent calls to the *DosSetNmpHandState*( ) function.

The parameters are as follows:

handle Specifies the handle of a named pipe instance.

pmode Points to an unsigned short where the pipe open mode for the named pipe instance (as defined below) is returned.

Manifest	Bit Mask	Meaning
NP_ICOUNT	0x00FF	Instance count (maximum number of instances allowed to be open concurrently for the named pipe). If 0xFF, an implementation-dependent maximum number of instances can be created. The value 0 is reserved.
NP_RMESG	0x0100	Pipe read mode. If 0, the pipe instance can be read as a byte stream. If 1, the pipe instance can be read either as a byte or a message stream pipe.
NP_WMESG	0x0400	Pipe type. If 0, the pipe is in byte stream mode. If 1, the pipe is in message mode.
NP_SERVER	0x4000	If 1, serving end of pipe. If 0, client end of the pipe.
NP_NBLK	0x8000	Indicates the blocking mode. If 0 (blocking mode), all reads and writes block when no data is available.

	Reads from a named pipe block until at least some data is available. Writes to a named pipe block until all data is written. If 1 (non-blocking mode), all reads and writes return immediately when no data is available. The blocking mode can be reset by calling the *DosSetNmpHandState*( ) function.
-	All other bits are undefined.

RETURN VALUE

Manifest	**Meaning**
[NERR_Success]	No errors encountered.
[ERROR_BAD_PIPE]	If client side made call and handle does not refer to a named pipe instance.
[ERROR_INVALID_HANDLE]	Invalid handle specified.
[ERROR_PIPE_NOT_CONNECTED]	If client side made call and the named pipe is currently not in the NP_CONNECTED state.

SEE ALSO

DosQFHandState( )	Retrieve information about a named pipe handle.
DosSetNmpHandState( )	Modify the handle state of a named pipe instance.

NAME

DosQNmPipeInfo( ) — retrieves information about the sizes of a particular named pipe's incoming and outgoing buffers and how many instances are available.

SYNOPSIS

```
#include <lm/nmpipe.h>
#include <lm/lmerror.h>

unsigned
DosQNmPipeInfo(
               handle,
               level,
               buf,
               buflen
              )

unsigned handle;
short      level;
char *     buf;
unsigned short buflen;
```

DESCRIPTION

The parameters are as follows:

handle Specifies the handle of a named pipe instance.

level Specifies the level of detail of information to be returned in *buf*.

buf Points to the data area appropriate for the level of information requested.

buflen Specifies the size in bytes of *buf*.

If the size of the buffer supplied by the caller is too small for the information to be returned, as much information as fits is moved into the buffer and the error [ERROR_BUFFER_OVERFLOW] is returned. The system discards the additional information.

The only valid value for *level* is 1. At this level, on success the information returned is in the following structure:

```
struct npi_data1 {
                  unsigned short npi_obuflen;
                  unsigned short npi_ibuflen;
                  unsigned char  npi_maxicnt;
                  unsigned char  npi_curicnt;
                  unsigned char  npi_namlen;
                           char  npi_name[1];
                 };
```

where:

npi_obuflen Size in bytes of the named pipe's outgoing I/O buffer.

npi_ibuflen Size in bytes of the named pipe's incoming I/O buffer.

npi_maxicnt How many instances are allowed for the named pipe.

npi_curicnt How many instances of the named pipe are open.

npi_namlen Length of *npi_name*.

npi_name A variable-length ASCIIZ string containing the pipe's name, in the form **\PIPE\name**.

RETURN VALUE

Manifest	Meaning
[NERR_Success]	No errors encountered.
[ERROR_BAD_PIPE]	If client side made call and handle does not refer to a named pipe instance.
[ERROR_BUFFER_OVERFLOW]	Buffer too small.
[ERROR_INVALID_HANDLE]	Invalid handle specified.
[ERROR_INVALID_LEVEL]	Invalid *level* parameter.

SEE ALSO

DosMakeNmPipe( )	Create a named pipe, or new instance of existing named pipe, and return handle.
DosQNmpHandState( )	Retrieve the current state of named pipe handle instance.

NAME

DosRead( ) — reads data from a named pipe instance.

SYNOPSIS

```
#include <lm/nmpipe.h>
#include <lm/lmerror.h>

unsigned
DosRead(
        handle,
        buf,
        buflen,
        bytesread
       )

unsigned handle;
char * buf;
unsigned short buflen;
unsigned short * bytesread;
```

DESCRIPTION

The parameters are as follows:

handle Specifies the handle of the named pipe instance.

buf Points to the buffer where data is read into.

buflen Specifies the size (in bytes) of the buffer.

bytesread Points to an unsigned short integer telling how many bytes were read.

If *DosRead*( ) returns successfully with *bytesread* set to 0, then no data is available because the pipe was closed at the other end.

The behaviour of *DosRead*( ) is controlled by the state of the NP_RMESG, NP_WMESG and NP_NBLK bits of the named pipe handle state (see *DosQNmpHandState*( ) on page 50).

WMESG	RMESG	Behaviour
0	0/1	All available data, up to the size specified by *buflen*, is returned in *buf*.
1	0	Only the data in the messages is returned, as if it had been written to a byte-mode (NP_WMESG=0) pipe. Message headers are discarded.
1	1	If *buf* is larger than the message, the message is read and *DosRead*( ) sets *bytesread* to the number of bytes read from the named pipe. If *buf* is smaller than the message, *DosRead*( ) reads as many bytes as the buffer can store, sets *bytesread* to the number of stored bytes, and returns the [ERROR_MORE_DATA] error code. In this case, the next call to *DosRead*( ) picks up where this one left off, reading the remaining portion of the message.

In blocking mode (NP_NBLK=0), a read waits until any data is available. If NP_WMESG=NP_RMESG=1, a blocking-mode read waits until an entire message is available or until the current message fills the buffer.

In non-blocking mode (NP_NBLK=1), a read immediately returns an [ERROR_NO_DATA] error if no data is available. If a read returns the [ERROR_MORE_DATA] error, a message was received that is too large to fit in *buf*; as much of the message as would fit is returned, and another read should be initiated to read the remaining data. (Since this condition can only occur when NP_WMESG=NP_RMESG=1, the first read would not return until the entire message was received; hence, the subsequent read should return more data immediately.)

RETURN VALUE

Manifest	**Meaning**
[NERR_Success]	No errors encountered.
[ERROR_ACCESS_DENIED]	Access not allowed.
[ERROR_BROKEN_PIPE]	The pipe has been closed or has no reader.
[ERROR_MORE_DATA]	More data available, buffer too small.
[ERROR_NO_DATA]	Data unavailable on non-blocking read.
[ERROR_PIPE_BUSY]	The specified pipe is busy.
[ERROR_PIPE_NOT_CONNECTED]	Named pipe is currently not in the NP_CONNECTED state.

SEE ALSO

DosPeekNmPipe( )	Read data from a named pipe instance without removing it.

NAME

DosSetFHandState( ) — modifies some of the open modes for a named pipe instance in NP_CONNECTED state.

SYNOPSIS

```
#include <lm/nmpipe.h>
#include <lm/lmerror.h>

unsigned
DosSetFHandState(
                 handle,
                 handlestate
                )

unsigned handle;
unsigned short handlestate;
```

DESCRIPTION

The parameters are as follows:

handle Specifies the handle of a named pipe instance.

handlestate A bitmap specifying the state of a named pipe handle, defined as follows:

Manifest	Bit Mask	Meaning
NP_INHERITANCE	0x0080	If 0, then spawned processes inherit the named pipe handle. If 1, then spawned processes do not inherit the handle.
NP_WB	0x4000	If 0, write-behind to remote pipes is allowed. If 1, write-behind is not allowed. Behaviour for this mode is implementation-defined.
-		All other bits are undefined.

RETURN VALUE

Manifest	Meaning
[NERR_Success]	No errors encountered.
[ERROR_BAD_PIPE]	Handle does not refer to a named pipe instance.
[ERROR_INVALID_PARAMETER]	Invalid parameter specified.
[ERROR_PIPE_NOT_CONNECTED]	Named pipe is currently not in the NP_CONNECTED state.

SEE ALSO

DosQNmpHandState( )	Retrieve the current state of named pipe handle instance.
DosQFHandState( )	Retrieve information about a named pipe handle.
LmForkNmPipe( )	Fork a CAE process that inherits named pipe handles.

NAME

DosSetNmpHandState( ) — modifies the state of the read mode and blocking mode of a named pipe instance in NP_CONNECTED state.

SYNOPSIS

```
#include <lm/nmpipe.h>
#include <lm/lmerror.h>

unsigned
DosSetNmpHandState(
                   handle,
                   handlestate
                  )

unsigned handle;
unsigned short handlestate;
```

DESCRIPTION

The parameters are as follows:

handle Specifies the handle of a named pipe instance.

handlestate A bitmap that sets the mode of the handle, defined as follows:

Manifest	Bit Mask	Meaning
NP_RMESG	0x0100	If 0, the pipe instance is read as a byte stream. If 1, the pipe instance is read either as a message stream or a byte stream.
NP_NBLK	0x8000	If 0, all reads and writes are blocked when no data is available. If 1, all reads and writes return immediately (non-blocking) when no data is available.
-		All other bits are undefined.

The *handlestate* bit settings correspond to those in the *DosMakeNmPipe*( ) function's *pmode* argument. *DosQNmpHandState*( ) retrieves information on these settings.

RETURN VALUE

Manifest	Meaning
[NERR_Success]	No errors encountered.
[ERROR_BAD_PIPE]	Handle does not refer to a named pipe instance.
[ERROR_INVALID_HANDLE]	Invalid handle specified.
[ERROR_INVALID_PARAMETER]	Invalid parameter specified.
[ERROR_PIPE_NOT_CONNECTED]	Named pipe is currently not in the NP_CONNECTED state.

SEE ALSO

DosQNmpHandState( )	Retrieve the current state of named pipe handle instance.
DosQFHandState( )	Retrieve information about a named pipe handle.

NAME

DosTransactNmPipe( ) — writes a message to and then reads a message from a named pipe instance.

SYNOPSIS

```
#include <lm/nmpipe.h>
#include <lm/lmerror.h>

unsigned
DosTransactNmPipe(
                  handle,
                  writebuf,
                  writelen,
                  readbuf,
                  readlen,
                  bytesread
                 )

unsigned handle;
char *writebuf;
unsigned short writelen;
char *readbuf;
unsigned short readlen;
unsigned short *bytesread;
```

DESCRIPTION

DosTransactNmPipe( ) provides a mechanism for two processes to complete a transaction over a message named pipe instance. *DosTransactNmPipe*( ) fails if the named pipe instance is not in message mode or is in message mode and is not a bidirectional named pipe.

A named pipe's blocking state has no effect on *DosTransactNmPipe*( ). The function does not return until a message is stored into *readbuf*, even in non-blocking mode. If *readbuf* is too small to hold the entire returned message, *DosTransactNmPipe*( ) returns an [ERROR_MORE_DATA] error, filling *readbuf* with as many bytes as fit. The remaining data can be obtained with a call to *DosRead*( ).

When using *DosTransactNmPipe*( ) it is assumed the programs using the interface are working in a *request/response* mode. Therefore it may be useful to place a call to *DosBufReset*( ) prior to the first use of *DosTransactNmPipe*( ) to ensure that no data is in the named pipe prior to starting the request/response sequence.

The parameters are as follows:

handle Specifies the file descriptor of a named pipe instance.

writebuf Points to the buffer containing data to write to the pipe.

writelen Specifies the size (in bytes) of *writebuf*.

readbuf Points to the buffer containing returned data.

readlen Specifies the size (in bytes) of *readbuf*.

bytesread Points to an unsigned short integer telling how many bytes were actually read from the pipe instance.

RETURN VALUE

Manifest	**Meaning**

[NERR_Success]	No errors encountered.
[ERROR_BAD_FORMAT]	Pipe is a byte stream pipe or is not bidirectional.
[ERROR_BAD_PIPE]	Handle does not refer to a named pipe instance.
[ERROR_BROKEN_PIPE]	The pipe has been closed or has no reader.
[ERROR_INVALID_HANDLE]	Invalid handle specified.
[ERROR_MORE_DATA]	More data available, buffer too small.
[ERROR_PIPE_BUSY]	The specified pipe is busy.
[ERROR_PIPE_NOT_CONNECTED]	Named pipe is currently not in the NP_CONNECTED state.

SEE ALSO

DosBufReset( )	Flush a named pipe instance.
DosRead( )	Read data from a named pipe instance.
DosWrite( )	Write data to a named pipe instance.

NAME

DosWaitNmPipe( ) — waits for an instance of the named pipe to become available.

SYNOPSIS

```
#include <lm/lmerror.h>
#include <lm/nmpipe.h>

unsigned
DosWaitNmPipe(
              name,
              timeout
             )

char * name;
unsigned long timeout;
```

DESCRIPTION

The function *DosWaitNmPipe*( ) can be used when *DosOpen*( ) returns with the error code [ERROR_PIPE_BUSY] to allow the process to wait for an instance of the named pipe to become available. If more than one process is waiting for the named pipe to become available, LMX gives the next available named pipe instance to the process that has been waiting the longest.

The parameters are as follows:

name Points to an ASCIIZ string containing the name of a pipe in the format **\PIPE\name** for a local pipe or **\\LMXservername\PIPE\name** for a remote pipe.

timeout Specifies the amount of time in milliseconds the LMX server should wait for the named pipe to become available. If the value is –1, the default time (set by *DosMakeNmPipe*( )) is used. A value of 0 waits indefinitely.

RETURN VALUE

Manifest	**Meaning**
[NERR_Success]	No errors encountered.
[ERROR_BAD_PIPE]	Handle does not refer to a named pipe instance.
[ERROR_PATH_NOT_FOUND]	Unknown pathname specified.
[ERROR_INTERRUPT]	System call interrupted.
[ERROR_SEM_TIMEOUT]	Timeout occurred.

SEE ALSO

DosMakeNmPipe( )	Create a named pipe, or new instance of existing named pipe, and return handle.
DosOpen( )	Open the client end of a named pipe and return its handle.

NAME

DosWrite( ) — writes data to a named pipe instance.

SYNOPSIS

```
#include <lm/nmpipe.h>
#include <lm/lmerror.h>

unsigned
DosWrite(
         handle,
         buf,
         buflen,
         byteswritten
        )

unsigned handle;
char *buf;
unsigned short buflen;
unsigned short *byteswritten;
```

DESCRIPTION

If the named pipe is in message mode, each call to *DosWrite*( ) sends a single *message* - which in this case means the contents of the buffer passed during a call to *DosWrite*( ). A message header is automatically prepended to the data that is written. Data may be lost in a *DosWrite*( ) and *DosClose*( ) sequence. To ensure delivery of data a call to *DosBufReset*( ) should be made prior to calling *DosClose*( ).

The parameters are as follows:

handle Specifies the handle of a named pipe instance.

buf Points to the data that is to be written to the named pipe instance.

buflen Specifies the size (in bytes) of the data to write.

byteswritten Points to an unsigned short indicating how many bytes of data were written to the pipe instance. If this value is less than the number requested, it indicates that either the amount of data that the named pipe queues has reached its maximum or that there is a lack of resources to hold the data at the present time.

RETURN VALUE

Manifest	**Meaning**
[NERR_Success]	No errors encountered.
[ERROR_ACCESS_DENIED]	Access not allowed.
[ERROR_BROKEN_PIPE]	The pipe has been closed or has no reader.
[ERROR_PIPE_BUSY]	The specified pipe is busy or there is insufficient buffer space to complete the write.
[ERROR_PIPE_NOT_CONNECTED]	Named pipe is currently not in the NP_CONNECTED state.

SEE ALSO

DosBufReset( )	Flush a named pipe instance.
DosRead( )	Read data from a named pipe instance.

NOTES

Some books for the OS/2 system calls refer to the parameters *buflen* and *byteswritten* as being integers (see the referenced Waite Group document). On some implementations this means that the return of the *byteswritten* information may destroy some data if the memory area if *byteswritten* is not large enough to hold an integer.

NAME

LmForkNmPipe( ) — used by a CAE application to create a child process.

SYNOPSIS

```
#include <errno.h>
#include <lm/nmpipe.h>
#include <lm/lmerror.h>

int
LmForkNmPipe( )
```

DESCRIPTION

The child process inherits the parent named pipe handles that the parent allows to be inherited. Although some implementations of LMX support the inheritance of named pipe handles when the *fork*( ) is used, *LmForkNmPipe*( ) should be used in place of the *fork*( ) system call to ensure portability.

This call is designed for LMX implementations which do not support inheritance of named pipe handles across the *fork*( ) system call.

RETURN VALUE

LmForkNmPipe( ) sets its return value in the same way the *fork*( ) system call does. That is:

Value	Meaning
0	This value is returned to the child process.
–1	This value indicates that an error has occurred. In this case, **errno** is set to the appropriate CAE error.
pid	*pid* is the process ID of the child process. This value is returned to the parent process.

NOTES

It is possible that in attempting the *fork*( ) operation the LMX maximum value for named pipe handles may be exceeded. If this occurs, a –1 is returned and **errno** is set to [EAGAIN]. All other failures mean the *fork*( ) system call failed and **errno** is set by that failure.

SEE ALSO

DosMakeNmPipe( )	Create a named pipe, or new instance of existing named pipe, and return handle.
fork	Create a new process.
errno.h	**X/Open Portability Guide**, **Issue 3**, **Volume 2**, **XSI System Interface and Headers**.

3.5 Messaging

The Messaging API provides access to the simple NetBIOS message service provided by LMX. This form of messaging differs from that provided by the mailslot service in that simple messages can only be sent to a particular workstation. Generally, there is only one simple message delivery point on a single workstation. (There is some provision for forwarding of messages to a message server, but that server is still receiving all messages that were sent to the forwarded workstation through one delivery point.)

Messaging API Definition

The following page gives the Messaging API definition.

NAME

NetMessageBufferSend( ) — sends a message to DOS or OS/2 systems.

SYNOPSIS

```
#include <stdio.h>
#include <lm/message.h>
#include <lm/lmerror.h>

unsigned
NetMessageBufferSend(
                     messenger_name,
                     name,
                     buf,
                     buf_len
                    )

char * messenger_name;
char * name;
char * buf;
unsigned short buf_len;
```

DESCRIPTION

NetMessageBufferSend( ) sends a message to one or more users or applications on DOS or OS/2 systems, not on other CAE systems. A server may send a buffer without necessarily being able to receive any data itself; in this case, *name* cannot point to the name of that server.

Not all systems are capable of forwarding the *NetMessageBufferSend*( ) request to a different sender; on those systems, *messenger_name* must be NULL.

The maximum message size is a configuration-dependent parameter of the receiving system and is no greater than 65535.

The parameters are as follows:

messenger_name
: Points to an ASCIIZ string containing the name of the remote server from which the message should be sent. A NULL pointer indicates the message should be sent from the local server. The maximum length of the name is 15 characters.

name
: Points to an ASCIIZ string specifying the registered user or application to which the buffer should be sent. The buffer is sent to all registered users and applications if *name* points to the character string "*".

buf
: Points to the message to be sent. The message need not be terminated by a zero byte and may contain zero bytes.

buf_len
: Specifies the size in bytes of *buf*.

RETURN VALUE

Manifest	**Meaning**
[NERR_Success]	No errors encountered.
[NERR_NameNotFound]	The name does not exist as a message name at any system.
[NERR_BadReceive]	A communication error occurred while sending the message.
[NERR_TruncatedBroadcast]	The broadcast message was truncated, only the first 128 bytes were sent.

[ERROR_NAME_NOT_FOUND]	The remote computer name was not found.
[ERROR_REM_NOT_LIST]	This remote computer is not listening.

SEE ALSO

<stdio.h>	**X/Open Portability Guide**, **Issue 3**, **Volume 2**, **XSI System Interface and Headers**.

Chapter 4

Transmission Analysis

This chapter addresses the relationship between the LMX API parameters and the SMB requests that are generated by the API. From this information it is possible to see the relationship between the API and the SMB requests, and to understand any mapping or transformation that may occur on any of the API parameters. This chapter does not document any particular implementation of the API to SMB mapping; it is an example of what can be done. This description focuses on the logical request/response sequence without describing the issues regarding secondary *SMBtrans* requests.

4.1 Mailslots

In this section, a few assumptions are made as described in Section 4.1.1.

4.1.1 Introduction

For the mailslot APIs that require an LMX session (that is, support for Class 1 mailslots), it is assumed that the session with the server is established and operational. The construction of an LMX session requires the following sequence:

Client	Server
Establish LMX session to server system.	
Send an *SMBnegprot*.	
	Receipt of negotiation and transmission of reply.
Verify that the extended dialect was chosen. If not, the sequence is aborted.	
Send an *SMBsesssetupX* if user-level security is negotiated.	
	In user-level security validate that the user name and password provided are correct.
If no validation error, send an *SMBtcon* for the IPC$ share.	
	Validate that the user has access to the IPC$ share.
If no error, the session is established and other SMB requests can follow as described below.	

If an LMX session is already established at the time a mailslot API is called, a connection to the IPC$ resource is required. The sequence to do this is as follows:

Client	Server
If the extended dialect was not negotiated, return [ERROR_PATH_NOT_FOUND].	
Send an *SMBtcon* for the IPC$ resource.	
	Validate that the user has access to the IPC$ resource.
If no error, the connection to IPC$ is established and other IPC SMB requests can be exchanged.	

Mapping of errors during this process to the caller is as follows:

Virtual circuit connection failure	ERROR_PATH_NOT_FOUND
Negotiate failure or wrong level	ERROR_PATH_NOT_FOUND
Session Setup failure	ERROR_PATH_NOT_FOUND
Insufficient rights to access IPC$ resource	ERROR_ACCESS_DENIED
Other SMB failures	Error code as mapped in Chapter 5 on page 103.

To support Class 2 mailslots the LMX server needs to support NetBIOS datagrams. Without establishing an LMX session directly, a NetBIOS datagram containing the *SMBtrans* request is sent. Therefore, LMX servers listen for incoming NetBIOS datagrams on the same NetBIOS name as for the LMX session support. Since Class 2 mailslots are not guaranteed, the responses to the *SMBtrans* request need not be sent by the receiving system.

4.1.2 DosDeleteMailslot( )

Invocation

```
DosDeleteMailslot(handle)
```

Processing

DosDeleteMailslot( ) removes the mailslot name from the system area reserved to maintain mailslot names.

Requests Generated

None.

4.1.3 DosMailslotInfo()

Invocation

```
DosMailslotInfo(
                handle,
                messagesize,
                mailslotsize,
                nextsize,
                nextpriority,
                msgcount
               )
```

Processing

DosMailslotInfo( ) simply returns the information concerning the mailslot specified by the handle. This call is only possible on the server side of mailslot processing.

Requests Generated

None.

4.1.4 DosMakeMailslot( )

Invocation

```
DosMakeMailslot(
                name,
                messagesize,
                mailslotsize,
                handle
               )
```

Processing

DosMakeMailslot( ) verifies the validity and uniqueness of the mailslot name. If the name is valid then it is stored for use when servicing is required.

Requests Generated

None.

4.1.5 DosPeekMailslot()

Invocation

```
DosPeekMailslot(
                handle,
                buf,
                bytesread,
                nextsize,
                nextpriority
               )
```

Processing

DosPeekMailslot( ) returns the next message on the mailslot if it is present without removing it from the message queue.

Requests Generated

None.

4.1.6 DosReadMailslot()

Invocation

```
DosReadMailslot(
                handle,
                buf,
                bytesread,
                nextsize,
                nextpriority,
                timeout
               )
```

Processing

DosReadMailslot( ) only returns the next message on the mailslot if a message is present. The function waits for data to arrive for the specified timeout if no data is present.

Requests Generated

None.

4.1.7 DosWriteMailslot()

Invocation

```
DosWriteMailslot(
                 name,
                 message,
                 size,
                 priority,
                 class,
                 timeout
                )
```

Processing

For both Class 1 and Class 2 mailslots this function uses the LMX server name given in the **\\computername\MAILSLOT\name** path pointed to by *name*. The remainder of the **\MAILSLOT\name** is moved, and zero-terminated, into the *smb_name*[ ] field of the *SMBtrans* request. The *size* parameter is used to determine the amount of user supplied data to move in the *smb_data*[ ] portion of the *SMBtrans*(*WriteMailslot*) request. The *timeout* parameter is transferred directly into the *smb_timeout* field of the *SMBtrans* request. The *smb_setup*[0] field is set to the command code for WriteMailslot. *priority* is copied directly into the *smb_setup*[1] field and *class* is copied directly into the *smb_setup*[2] field of the *SMBtrans* request.

The setting of the *smb_flags* field is largely implementation-dependent, but can be used in the following manner. For Class 2 mailslots bit 1 of this field can be set to indicate that no response is required in order to decrease the overhead of both the server and client machines. Class 1 mailslots must use an existing session or create a new session if none exists on behalf of the client system user and send the request over the session. Using this approach there is no need for the response bit to be set for a response since the session guarantees the delivery of the request. Alternatively, the application may choose to set bit 1 to indicate a response is required.

The setting of bit 0 to indicate the automatic disconnection of a TID supplied in *smb_tid* could be used if the client and server implementation of the SMB allow for the connection of a generic mailslot service.

Requests Generated

Class 2: *SMBtrans*(*WriteMailslot*)

Class 1: If no connection to IPC$ exists:

Refer to Section 4.1.1 on page 67.

After the connection exists:

SMBtrans(*WriteMailslot*)

If the connection to the server did not exist prior to the transaction, it may be disconnected at this time.

4.2 Named Pipes

In this section, a few assumptions are made as described in Section 4.2.1.

4.2.1 Introduction

For each of the named pipe APIs that have handles and require network traffic, it is assumed that the session with the server is established and operational. In a few APIs it is possible that the session is not established, therefore a comment is made in the **Requests Generated** section that a session must be established. The construction of an LMX session requires the following sequence:

Client	**Server**
Establish LMX session to server system.	
Send an *SMBnegprot*.	
	Receipt of negotiation and transmission of reply.
Verify that the extended dialect was chosen. If not, the sequence is aborted.	
Send an *SMBsesssetupX*.	
	If in user-level security validate that the user name and password provided are correct.
If no validation error, send an *SMBtcon* for the IPC$ share.	
	Validate that the user has access to the IPC$ share.
If no error, the session is established and other SMB requests can follow as described below.	

During this process, mapping of errors to be returned to the caller is as follows:

Virtual circuit connection failure	ERROR_PATH_NOT_FOUND
Negotiate failure or wrong level	ERROR_PATH_NOT_FOUND
Session Setup failure	ERROR_PATH_NOT_FOUND
Insufficient rights to access IPC$ resource	ERROR_ACCESS_DENIED
Other SMB failures	Error code as mapped in Chapter 5 on page 103.

Due to considerations of space and performance trade-offs, there are different ways in which some of the named pipe APIs may work. This section touches on each of these approaches. There are scenario tables listed in the **Requests Generated** section for the APIs that have multiple implementations.

4.2.2 DosBufReset( )

Invocation

```
DosBufReset(handle)
```

Processing

When called on the server side of the pipe no protocol is generated. The processing for the server code is to cause the caller's execution path to wait until the client side has read all of the data cached in the named pipe.

When called on the client side all cached data for the file handle is written using *SMBwrite* requests. After completion of the writes, an *SMBflush* is sent to wait for all of the data to be read on the server side. *DosBufReset*( ) waits for an implementation-defined amount of time, in seconds, for the data to be read from the named pipe. The suggested default is to have no timeout.

The writing of the cached data sequence can be done with the *SMBwriteX* protocol.

Requests Generated

Scenario 1	Scenario 2
SMBwrite	*SMBwriteX*
SMBflush	*SMBflush*

4.2.3 DosCallNmPipe()

Invocation

```
DosCallNmPipe(
              name,
              writebuf,
              writelen,
              readbuf,
              readlen,
              bytesread,
              timeout
             )
```

Processing

DosCallNmPipe( ) can be implemented as a sequence of calls starting from the opening of the named pipe via *DosOpen*( ), sending and receiving replies via *DosTransactNmPipe*( ), and the closing of the named pipe handle via *DosClose*( ). The parameters given to the routine map to parameters for these functions.

A more optimal approach is to use the *SMBtrans* protocol with the *smb_setup*[0] field set to *CallNmPipe*. The server name is given in the **\\LMXservername\PIPE\name** path pointed to by *name*. The remainder of the **\PIPE\name** is moved, and zero-terminated, into the *smb_name*[ ] field of the *SMBtrans* request. The *timeout* parameter is copied into the *smb_timeout* field of the *SMBtrans* request. The data bytes to write in the named pipe are moved into the *smb_data*[ ] field from the area referenced by the *writebuf* parameter for the size specified by the *writelen* parameter. When the response is received the data is moved from the *smb_data*[ ] field of the *SMBtrans* request into the buffer referenced by the *readbuf* parameter up to the maximum of the size specified by the *readlen* parameter and the number of data bytes received as indicated in *smb_tdrcnt*. This size of the data moved is stored in the unsigned short location pointed to by the *bytesread* parameter.

Request mapping:

API parameter	action	SMB location
name	string copied to	*smb_name*
writebuf	memory copied for *writelen* bytes to	*smb_data*
writelen	stored in	*smb_tdscnt*
readlen	stored in	*smb_mdrcnt*
timeout	stored in	*smb_timeout*

Response mapping:

SMB location	action	API parameter
smb_data	memory copied for *smb_tdrcnt* bytes to	*readbuf*
smb_tdrcnt	stored in	*bytesread*

If the *readlen* value is less than the amount of data to be returned, the data up to *readlen* size is copied into the *readbuf* data area and the error code [ERROR_MORE_DATA] is returned to the caller. Since *DosCallNmPipe*( ) closes the pipe after the action, the additional data is lost.

Requests Generated

If no connection to IPC$ exists:

Refer to Section 4.2.1 on page 75.

After the connection exists:

Scenario 1	Scenario 2	Scenario 3
SMBopenX	*SMBopenX*	*SMBtrans(CallNmPipe)*
SMBwrite	*SMBwriteX*	
SMBread	*SMBreadX*	
SMBclose	*SMBclose*	

4.2.4 DosClose()

Invocation

```
DosClose(handle)
```

Processing

When called on the server side of the named pipe, *DosClose*( ) places the named pipe instance into an NP_CLOSING state. This state means all subsequent server calls to read or write on this named pipe instance return with errors, yet all data that has been queued in the pipe remains until the client side has performed the reads necessary to receive the data or closes the named pipe. After the data has been flushed from the named pipe instance the resources associated with the instance are completely freed. Any use of the named pipe handle by the server at this time results in an error.

When called on the client side all cached data is written using *SMBwrite* or *SMBwriteX* requests. After completion of the writes, the named pipe is closed with *SMBclose.*

Requests Generated

If there is buffered data:

SMBwrite or *SMBwriteX*

After buffered data is flushed:

SMBclose

4.2.5 DosConnectNmPipe( )

Invocation

```
DosConnectNmPipe(handle)
```

Processing

DosConnectNmPipe( ) is a server-side call only. If the blocking mode is selected, *DosConnectNmPipe*( ) causes the caller to wait until the specified named pipe instance indicated by the parameter *handle* is opened. In non-blocking mode this call is necessary to change the state of the named pipe instance from NP_DISCONNECTED to NP_LISTEN such that a client *DosOpen*( ) succeeds.

Requests Generated

None.

4.2.6 DosDisconnectNmPipe( )

Invocation

```
DosDisconnectNmPipe(handle)
```

Processing

DosDisconnectNmPipe( ) is a server-side call that marks the network file handle for the named pipe referenced via the parameter *handle* as invalid. This has a similar action as for *DosClose*( ) on the server side of the named pipe except that after the client has read all of the cached data the named pipe instance returns to the NP_DISCONNECTED state and it is legal for the server side to issue the *DosConnectNmPipe*( ) function call.

Requests Generated

None.

4.2.7 DosDupHandle( )

Invocation

```
DosDupHandle(
             oldhandle,
             newhandle
            )
```

Processing

If *newhandle* is a valid handle to a named pipe a *DosClose*( ) is performed on the handle, otherwise no SMB is sent. *DosDupHandle*( ) then performs the duplication function.

Requests Generated

None.

4.2.8 DosMakeNmPipe( )

Invocation

```
DosMakeNmPipe(
              name,
              handle,
              omode,
              pmode,
              outsize,
              insize,
              timeout
             )
```

Processing

DosMakeNmPipe( ) stores the pipe name and associated information in such a way that it can be accessed by the LMX implementation to respond to *SMBopenX* protocols containing the name. The initial state of this instance is set to NP_DISCONNECTED. If the *timeout* value is defaulted (that is, equal to 0), the suggested timeout default is 50 milliseconds.

Requests Generated

None.

4.2.9 DosOpen( )

Invocation

```
DosOpen(
        name,
        handle,
        action,
        size,
        attribute,
        openflag,
        omode,
        reserved
       )
```

Processing

This is a client-side call. *DosOpen*( ) uses the server name given in the **\\LMXservername\PIPE\name** path pointed to by *name*. The remainder of the **\PIPE\name** is moved, and zero-terminated, into the *smb_pathname*[ ] field of the *SMBopenX* request. The *size* parameter is copied into the *smb_size* field of the *SMBopenX* request. The *attribute* parameter is copied into the *smb_attr* field of the *SMBopenX* request. The *openflag* parameter is mapped into the *smb_ofun* field of the *SMBopenX* request. The *omode* parameter is copied into the *smb_mode* field of the *SMBopenX* request. The remaining *SMBopenX* information is filled out as directed by the SMB specification (see **Protocols for X/Open PC Interworking: SMB**). The *smb_time* field has no meaning on named pipes since they cannot be created by *DosOpen*( ). The *smb_timeout* may be ignored. In the response to the *SMBopenX* the client system uses the resource type field (*smb_type*) to recognise that the item opened is either a message-mode or stream-mode named pipe. For further information see Section 4.2.19 on page 97.

Request mapping:

API parameter	**action**	**SMB location**
name	string copied to	*smb_name*
size	stored in	*smb_size*
attribute	stored in	*smb_attr*
openflag	mapped into	*smb_ofun*
omode	stored in	*smb_mode*

Response mapping:

SMB location	**usage**
smb_fid	stored for later retrieval via *handle* returned.
smb_type	stored to indicate type of named pipe opened.
smb_action	stored in unsigned short located by *action* parameter.

If the *SMBopenX* succeeds, a value must be returned to the caller which acts as the handle to the named pipe. This value does not have to be the value returned in the *SMBopenX* request in the *smb_fid* field, it can be any value. But, the implementation must make a mapping between the value returned and the value received in the *SMBopenX* request.

Requests Generated

If no connection to IPC$ exists:

Refer to Section 4.2.1 on page 75.

After the connection exists:

SMBopenX

4.2.10 DosPeekNmPipe()

Invocation

```
DosPeekNmPipe(
              handle,
              buf,
              buflen,
              bytesread,
              bytesavail,
              status
             )
```

Processing

When called on the server side of the named pipe *DosPeekNmPipe*( ) checks for locally (located on the server) buffered data and returns the expected information based on this data.

When called on the client side of the named piped an *SMBtrans* with the transaction type set to *PeekNmPipe* is sent. The parameter *handle* is used to obtain the network file handle for the named pipe which is stored in the *smb_setup*[1] field of the *SMBtrans* request. The field *smb_dscnt* is set to the value of the parameter *buflen*. From the response, the data (if any) and the count of the data is returned in the data areas referenced by the *buf* and *bytesread* parameters. The two short unsigned integers referenced by *bytesavail* are filled in with the value from the *smb_param*[0] area of the *SMBtrans* response per the definition in the **Protocols for X/Open PC Interworking: SMB**. The unsigned short referenced by the parameter *status* is set to the value of the status information returned in the *smb_param*[2] field of the *SMBtrans* response per the description in the **Protocols for X/Open PC Interworking: SMB**.

Request mapping:

API parameter	**action**	**SMB location**
handle	used to derive value for	*smb_fid*
buflen	stored in	*smb_dscnt*

Response mapping:

SMB location	**action**	**API parameter**
smb_data[ ]	memory copied to	*buf*
smb_dscnt	copied to	*bytesread*
smb_param[0]	two words copied to	*bytesavail*
smb_param[2]	copied to	*status*

Requests Generated

SMBtrans(*PeekNmPipe*)

4.2.11 DosQFHandState( )

Invocation

```
DosQFHandState(
               handle,
               handlestate
              )
```

Processing

For both the server and client side of a named pipe *DosQFHandState*( ) gets the state information for the named pipe handle in the local system and returns it in the memory location pointed to by the parameter *handlestate*.

Requests Generated

None.

4.2.12 DosQNmPipeInfo( )

Invocation

```
DosQNmPipeInfo(
               handle,
               level,
               buf,
               buflen
              )
```

Processing

When called on the server side, no protocol is generated for this function. The function simply returns the information concerning the named pipe.

On the client side, the parameter *handle* is used to obtain the network file handle for the named pipe which is stored in the *smb_setup*[1] field of the *SMBtrans* request. The parameter *level* is copied into the *smb_param*[0] field of the *SMBtrans* request and the request is sent to the server machine. When a response is returned, the response data area, *smb_data*, contains the information to be returned to the caller in the data buffer referenced by the *buf* parameter. This information is transferred by the *DosQNmPipeInfo*( ) function into the provided data area up to the size specified by the parameter *buflen*. In the transfer of the data the client system must ensure that the prefix **\\<LMXservername>** of the remote server is added to the name of the pipe.

Requests Generated

SMBtrans(*QNmPipeInfo*)

4.2.13 DosQNmpHandState( )

Invocation

```
DosQNmpHandState(
                 handle,
                 pmode
                )
```

Processing

When called on the server side *DosQNmpHandState*( ) returns the server state for the named pipe. When called on the client side the *SMBtrans* protocol is generated. The only parameter mapping for the request protocol is the handle to the network file identifier which is stored in the *smb_setup*[1] field of the protocol. For the response, the pipe state information referenced in the *smb_param*[0] is stored in the unsigned short location pointed to by the parameter *pmode*.

Requests Generated

SMBtrans(*QNmpHandState*)

4.2.14 DosRead()

Invocation

```
DosRead(
        handle,
        buf,
        buflen,
        bytesread
       )
```

Processing

When called on the server side this function performs its action based on whether the named pipe is in blocking or non-blocking mode. In blocking mode *DosRead*( ) waits until data is received via a write request from the client side of the named pipe or the client closes the named pipe. In non-blocking mode *DosRead*( ) returns with the unsigned short referenced by the parameter *bytesread* set to zero if there is no data and the error [ERROR_NO_DATA] is returned.

When called on the client side *DosRead*( ) may generate an *SMBread*, or another read protocol for byte stream pipes. The parameter *handle* is used to obtain the network file handle for the named pipe. When the response to the *SMBread* is returned, the data received in the *smb_bcc*[ ] portion of the protocol is copied into the buffer area referenced by the parameter *buf* and the number of bytes received is stored in the unsigned short referenced by the parameter *bytesread*. If the pipe is in message mode then the error code returned with the data indicates whether the complete message has been returned. If the values for *smb_rcls* and *smb_err* (error class and error code) are [ERRDOS] and [ERRmoredata], respectively, then *DosRead*( ) must issue another read request until this error is no longer returned or the requested *buflen* amount has been satisfied. If the *buf* should become full before all data for the message has been read, the function returns [ERROR_MORE_DATA] to the caller. When no error is returned in the *SMBread* response, then the complete message has been returned and copied into the user buffer. Note that it is possible for the data length to be greater than the buffer supplied in the *DosRead*( ) call. In this case all of the data that fits in the buffer is returned and the error [ERROR_MORE_DATA] is returned to the caller. Depending on the caching scheme used by the client, or the amount of data expected in the response to the read, the client software may choose to use the *SMBreadX*, *SMBreadbraw* or the *SMBreadbmpx* protocol to perform the action of the *DosRead*( ) function.

Requests Generated

SMBread, *SMBreadX*, *SMBreadbmpx* or *SMBreadbraw*

4.2.15 DosSetFHandState( )

Invocation

```
DosSetFHandState(
                 handle,
                 handlestate
                )
```

Processing

For both the server and client side of a named pipe *DosSetFHandState*( ) sets the state information for the named pipe handle in the local system based on the parameter *handlestate*.

Requests Generated

None.

4.2.16 DosSetNmpHandState( )

Invocation

```
DosSetNmpHandState(
                    handle,
                    handlestate
                   )
```

Processing

When called on the server side *DosQNmpHandState*( ) sets the server state for the named pipe. When called on the client side the *SMBtrans* protocol is generated. The *handle* parameter is used to provide the network file identifier for the message and stored in *smb_setup*[1]. The *handlestate* parameter is stored in the *smb_param*[0]. For the response, the pipe state information in *smb_param*[0] is stored in the unsigned short location pointed to by the parameter *handlestate.*

Requests Generated

SMBtrans(*SetNmPHandState*)

4.2.17 DosTransactNmPipe( )

Invocation

```
DosTransactNmPipe(
                  handle,
                  writebuf,
                  writelen,
                  readbuf,
                  readlen,
                  bytesread
                 )
```

Processing

Prior to any functioning on either the server or client side, a check is made to ensure that the named pipe referenced by *handle* is a bidirectional message-mode named pipe. If this condition is not true, *DosTransactNmPipe*( ) returns [ERROR_BAD_FORMAT].

When invoked on the server side *DosTransactNmPipe*( ) works entirely on local queues. No SMB requests are generated.

DosTransactNmPipe( ) only operates on the client side of a named pipe when the named pipe is in message mode. *DosTransactNmPipe*( ) can generate two different SMB request sequences. The first is to send the data referenced by the parameter *writebuf* via an *SMBwrite* protocol and then issue an *SMBread* and return the response data in the *readbuf* buffer and a count of bytes in the unsigned short referenced by the *bytesread* parameter. If the request and/or response for the transaction does not fit in the negotiated message size for the LMX session, this sequence expands in the same manner as *DosWrite*( ) and *DosRead*( ).

The second sequence for this function is to generate the *SMBtrans* protocol with the function code *TransactNmPipe* in the *smb_setup*[0] field. The *handle* parameter is used to find the network file handle for the pipe and the data located in *writebuf* is copied into the *smb_data*[ ] portion of the *SMBtrans*(*TransactNmPipe*) request. The response data is returned in the *smb_data*[ ] portion of the *SMBtrans*(*TransactNmPipe*) response and copied back into the user buffer area referenced by the parameter *readbuf*.

If the *readlen* value is less than the amount of data to be returned, the data up to *readlen* size is copied into the *readbuf* data area and the error code [ERROR_MORE_DATA] is returned to the caller. The remaining data is to be buffered until the caller issues a function call to either read the data (that is, *DosRead*( )) or destroy the data (that is, *DosClose*( ), *DosDisconnectNmPipe*( )).

Request mapping (*SMBtrans*(*TransactNmPipe*) sequence):

API parameter	action	SMB location
handle	used to derive value for	*smb_setup*[1]
writebuf	memory copied for *writelen* bytes to	*smb_data*[ ]
readlen	copied to	*smb_mdrcnt*

Response mapping:

SMB location	action	API parameter
smb_data[ ] *smb_mdrcnt*	memory copied for *smb_tdrcnt* bytes to stored in	*readbuf* *bytesread*

Requests Generated

Scenario 1	Scenario 2	Scenario 3
SMBwrite *SMBread*	*SMBwriteX* *SMBreadX*	*SMBtrans*(*TransactNmPipe*)

4.2.18 DosWaitNmPipe()

Invocation

```
DosWaitNmPipe(
              name,
              timeout
             )
```

Processing

This is a client-side call. This routine generates an *SMBtrans* request. *DosWaitNmPipe*( ) uses the server name given in the **\\LMXservername\PIPE\name** path pointed to by *name* to indicate which NetBIOS named server is requested for this named pipe. The remainder of the **\PIPE\name** is moved, and zero-terminated, into the *smb_name*[ ] field of the *SMBtrans* request. The *timeout* parameter is moved into the *smb_timeout* and the message is sent.

Requests Generated

SMBtrans(*WaitNmPipe*)

4.2.19 DosWrite()

Invocation

```
DosWrite(
         handle,
         buf,
         buflen,
         byteswritten
        )
```

Processing

When called on the server side of the pipe, data written with *DosWrite*( ) is queued until the client system issues a read to receive the data. The queuing preserves the message boundary.

When called on the client side, the parameter *handle* is used to obtain the network file handle for the selected write protocol. If the named pipe is in byte stream mode then the *SMBwrite* or *SMBwritebraw* request can be used. If the named pipe is in message mode then the *SMBwriteX* protocol must be used in order to communicate to the server the length of the message via the start of message bit in the *smb_wmode* field of the *SMBwriteX* request. The data referenced by the parameter *buf* is moved into the *smb_bcc*[ ] portion of the *SMBwrite* request for *buflen* bytes. For the *SMBwriteX* and *SMBwritebraw* protocols the data is moved into the *smb_data*[ ] field. When the response to the write is received the count of bytes actually written is taken from the field *smb_count* in the selected SMB response and stored in the unsigned short referenced by the parameter *byteswritten*.

Requests Generated

Scenario 1	Scenario 2	Scenario 3	Scenario 4
SMBwrite	*SMBwriteX*	*SMBwritebraw*	*SMBwritebmpx*

4.2.20 LmForkNmPipe()

Invocation

`LmForkNmPipe()`

Processing

This function provides the semantics of XSI *fork*( ) with the addition of named pipe inheritance. There are no protocols generated.

Requests Generated

None.

4.3 Messaging

Since there is only one API for messaging, this section is in two parts:

- Section 4.3.1 discusses how the transmissions for messaging are set up.
- *NetMessageBufferSend*( ) on page 65 discusses the construction of the SMB requests and responses.

4.3.1 Introduction

The messaging service allows for three forms of transmission for the message:

- The first is a directed message which is to be sent from the caller of *NetMessageBufferSend*( ) to a single recipient of the message.
- The second is a message directed at a group of users within the network.
- The third is a message that is broadcast to any and all message recipients on the network; that is, a broadcast message.

Support for the second form (that is, message directed at a group) is implementation-dependent. If an LMX implementation does not support this form of messaging, the function *NetMessageBufferSend*( ) returns the value [NERR_NameNotFound].

These three forms of messaging take different approaches to delivering the message. For the first case, a directed message, *NetMessageBufferSend*( ) attempts to establish a NetBIOS session with the specified system if an existing LMX session is not present. Two attempts are made, the first being to a *forwarded name* (see Chapter 6 on page 119), and the second being the message name; the sixteenth byte is 0x05 and 0x03 respectively. If one of these NetBIOS sessions is established, the function proceeds with the construction and transmission of the messaging protocols. There is no requirement to issue the *SMBnegprot* or other LMX server session setup protocols for the NetBIOS session. If this new NetBIOS session is used, then it is destroyed after the message has been sent.

In the case of a message directed at a group of users within the network, *NetMessageBufferSend*( ) sends the message using NetBIOS group names (refer to Chapters 14 and 15 of the **Protocols for X/Open PC Interworking: SMB**). The NetBIOS group name is constructed from a *domain* name. Each LMX server implementation has an implementation-defined method of specifying this name.

For broadcast messages, NetBIOS broadcast datagrams are used to send the protocol block. (See Chapters 14 and 15 of **Protocols for X/Open PC Interworking: SMB**.)

4.3.2 NetMessageBufferSend( )

Invocation

```
NetMessageBufferSend(
                     messenger_name,
                     name,
                     buf,
                     buf_len
                    )
```

Processing

The actions taken by *NetMessageBufferSend*( ) key on the messaging name located by *name*.

If *name* points to the character string representing the NetBIOS name of a system on the network, then the function performs the directed form of messaging. For the directed form of networking *buf_len* is compared against the maximum length of a message packet (128 bytes). If the length is less than or equal to this value, a single *SMBsends* is constructed as follows:

API parameter	**action**	*SMBsends* **location**
messenger_name	stored as type 04 buffer in	*smb_orig*
name	stored as type 04 buffer in	*smb_dest*
buf	stored as type 01 buffer of size *buf_len* in	*smb_message*

If the length is greater than 128, the message is broken up into an *SMBsendstrt*, *SMBsendtxt* and *SMBsendend* sequence as follows:

API parameter	**action**	*SMBsendstrt* **location**
messenger_name	stored as type 04 buffer in	*smb_orig*
name	stored as type 04 buffer in	*smb_dest*

A message group ID is returned from the destination system in the *SMBsendstrt* response in the field *smb_grpid*. This is called *grpid* and is used in the next protocol requests. One or more *SMBsendtxt* requests are sent until *buf_len* bytes of data have been transmitted. The maximum number of bytes per message stored in *smb_message* for *SMBsendtxt* is 128.

API parameter	**action**	*SMBsendtxt* **location**
grpid from *SMBsendstrt*	stored in	*smb_grpid*
buf	stored as type 01 buffer of up to 128 bytes in	*smb_message*

For each *SMBsendtxt* request there is a response. The requestor must wait for each response before sending the next request. After *buf_len* bytes of data have been transmitted an *SMBsendend* is sent.

API parameter	**action**	*SMBsendend* **location**

grpid from *SMBsendstrt*	stored in	*smb_grpid*

NetMessageBufferSend( ) then waits for the response to *SMBsendend*.

If *name* points to a character string representing a NetBIOS group name or the character string "*", then the function is to send the supplied message as a broadcast to a group or the entire network. This is done using a single *SMBsendb* request as follows:

API parameter	**action**	*SMBsendb* **location**
messenger_name	stored as type 04 buffer in	*smb_orig*
name	stored as type 04 buffer in	*smb_dest*
buf	stored as type 01 buffer of size *buf_len* in	*smb_message*

If *buf_len* is greater then 128 bytes, the first 128 bytes are sent in the message and the return code [NERR_TruncatedBroadcast] is returned. There is no response to this request.

Requests Generated

Scenario 1	**Scenario 2**	**Scenario 3**
SMBsends	*SMBsendstrt* *SMBsendtxt* *SMBsendend*	*SMBsendb* (broadcast only)

Chapter 5

SMB Protocol Specification for Named Pipes and Mailslots

This chapter supersedes information given in Appendix B of the referenced document **Protocols for X/Open PC Interworking: SMB**. Many terms are used within this section which are defined in **Protocols for X/Open PC Interworking: SMB**.

5.1 Extended SMB Transaction Requests

The *SMBtrans* requests are used to perform operations that are not necessarily related to file or print sharing. The *SMBtrans* is a generic operation similar to a remote procedure call; each *SMBtrans* request identifies the function to be performed and the parameters and raw data on which the function should operate, and expects parameters and raw data to be returned in the response.

SMBtrans is more flexible than the core SMBs in that all uses of *SMBtrans* may involve multiple request or response messages, circumventing many size and length limitations imposed on core SMBs. Because of their more flexible argument structure, *SMBtrans* requests can be used to perform more complex operations than are supported by other extended SMBs.

SMBtrans is sometimes used in a different fashion from other SMB requests. It is not always used in a strictly client/server fashion; that is, the sender of an *SMBtrans* request is not always a file-sharing client; nor is the responder to such a request always a file server. For this reason, the terms *client* and *server* are not used in the descriptions in this chapter; instead, *requester* and *responder* are used.

For the extended dialects, *SMBtrans* requests are used to provide all or part of the following services:

1. mailslot IPC
2. named pipe IPC
3. remote server administration

Future dialects may provide additional uses for *SMBtrans*.

5.2 SMBtrans Structure and Flow

Because of the flexibility supported, *SMBtrans* actually consists of four protocol elements. The set of all protocol elements sent to perform a particular function is referred to as a *transaction*.

1. The Primary Request is used to initiate a particular transaction.
2. The Secondary Request is used to continue a transaction started with a primary request; it is only used if the data required did not fit completely within the primary.
3. The Interim Response is used only if all the data required for the transaction did not fit within the primary request and the responder is expecting one or more secondary requests.
4. The Final Response is used to convey the actual reply to the request; one or more of these may be sent.

The TID, PID, UID and MID (reference **Protocols for X/Open PC Interworking: SMB**) must be the same for all requests and responses related to a single transaction.

5.2.1 Request Formats

Primary Request		**Secondary Request**	
Field Name	**Field Value**	**Field Name**	**Field Value**
smb_com	*SMBtrans*	*smb_com*	*SMBtranss*
smb_wct	14+*smb_suwcnt*	*smb_wct*	8
smb_vwv[0]	*smb_tpscnt*	*smb_vwv*[0]	*smb_tpscnt*
smb_vwv[1]	*smb_tdscnt*	*smb_vwv*[1]	*smb_tdscnt*
smb_vwv[2]	*smb_mprcnt*	*smb_vwv*[2]	*smb_pscnt*
smb_vwv[3]	*smb_mdrcnt*	*smb_vwv*[3]	*smb_psoff*
smb_vwv[4]	*smb_msrcnt*	*smb_vwv*[4]	*smb_psdisp*
smb_vwv[5]	*smb_flags*	*smb_vwv*[5]	*smb_dscnt*
smb_vwv[6-7]	*smb_timeout*	*smb_vwv*[6]	*smb_dsoff*
smb_vwv[8]	*smb_rsvd1*	*smb_vwv*[7]	*smb_dsdisp*
smb_vwv[9]	*smb_pscnt*	*smb_bcc*	
smb_vwv[10]	*smb_psoff*		*smb_param*
smb_vwv[11]	*smb_dscnt*		*smb_data*
smb_vwv[12]	*smb_dsoff*		
smb_vwv[13]	*smb_suwcnt*		
smb_vwv[14-]	*smb_setup*[ ]		
smb_bcc			
	smb_name		
	smb_param		
	smb_data		

Table 5-1 Transaction SMB Request Formats

smb_tpscnt A word containing the total number of parameter bytes being sent. This value may be revised downward in any or all secondary requests. The smallest value of *smb_tpscnt* sent during this transaction must equal the sum of all the *smb_pscnt* fields in all requests sent during the transaction.

smb_tdscnt A word containing the total number of data bytes being sent. This value may be revised downward in any or all secondary requests. The smallest value of *smb_tdscnt* sent during this transaction must equal the sum of all the *smb_dscnt* fields in all requests sent during the transaction.

smb_mprcnt A word containing the maximum number of parameter bytes the requester expects to be returned. The responder may not exceed this limit in its response.

smb_mdrcnt A word containing the maximum number of data bytes the requester expects to be returned. The responder may not exceed this limit in its response.

smb_msrcnt A word containing the maximum number of setup words the requester expects to be returned. The responder may not exceed this limit in its response. The value of *smb_msrcnt* must be less than or equal to 255 and is stored in the low-order byte of the word; the high-order byte is reserved and must be zero.

smb_flags A word containing flags altering the behaviour of the request. This flag field is distinct from that defined in **Protocols for X/Open PC Interworking: SMB**. The flags are:

Bit 0 If set, the TID on which this transaction was requested is closed after the transaction is completed.

Bit 1 If set, the transaction is *one way*; that is, no final response should be generated by the responder. An interim response, if required by the flow of the transaction, should be produced regardless of the setting of this bit.

Bits 2-15 Reserved; MBZ.

smb_timeout A 32-bit integer specifying the number of milliseconds to wait for completion of the requested operation before causing a timeout.

smb_rsvd1 A one-word reserved field which must be zero.

smb_pscnt A word indicating the number of parameter bytes being sent in this particular request; i.e., the size of *smb_param*.

smb_psoff A word giving the offset, in bytes, from the start of the SMB header to the beginning of the *smb_param* field. This permits *smb_param* to be preceded in the request by pad bytes to result in better alignment of the buffer.

smb_psdisp A word giving the displacement amongst all parameter bytes for this transaction of the parameter bytes contained in this request. This is used by the responder to correctly assemble all the parameter bytes received even if the requests were received out of sequence.

smb_dscnt A word indicating the number of data bytes being sent in this particular request; that is, the size of *smb_data.*

smb_dsoff A word giving the offset, in bytes, from the start of the SMB header to the beginning of the *smb_data* field. This permits *smb_data* to be preceded in the request by pad bytes to result in better alignment of the buffer.

smb_dsdisp A word giving the displacement amongst all data bytes for this transaction of the data bytes contained in this request. This is used by the responder to correctly assemble all the data bytes received even if the requests were received out of sequence.

smb_suwcnt A word containing the number of setup words sent in the primary request. This value must be less than or equal to 255 and is stored in the low-order byte of the word; the high-order word is reserved and must be zero.

smb_setup An array of words of setup data. The length of this array is given by *smb_swcnt* (above) and may be zero.

smb_name A null-terminated ASCIIZ string containing the transaction name. No pad bytes are permitted before this field; it must immediately follow the *smb_bcc* field.

smb_param A string of bytes, beginning at *smb_psoff* bytes into the request and containing *smb_pscnt* bytes. Padding may precede this field, as *smb_psdisp* points to its beginning; for the same reason, *smb_param* is not required to precede *smb_data* in each message (although it usually does).

smb_data A string of bytes, beginning at *smb_dsoff* bytes into the request and containing *smb_dscnt* bytes. Padding may precede this field, as *smb_dsdisp* points to its beginning; for the same reason, this field is not always required to follow *smb_param* (although it usually does).

5.2.2 Response Formats

Interim Response		**Final Response**	
Field Name	**Field Value**	**Field Name**	**Field Value**
smb_com	*SMBtrans*	*smb_com*	*SMBtrans*
smb_wct	0	*smb_wct*	10+*smb_suwcnt*
smb_bcc	0	*smb_vwv*[0]	*smb_tprcnt*
		smb_vwv[1]	*smb_tdrcnt*
		smb_vwv[2]	*smb_rsvd*
		smb_vwv[3]	*smb_prcnt*
		smb_vwv[4]	*smb_proff*
		smb_vwv[5]	*smb_prdisp*
		smb_vwv[6]	*smb_drcnt*
		smb_vwv[7]	*smb_droff*
		smb_vwv[8]	*smb_drdisp*
		smb_vwv[9]	*smb_suwcnt*
		smb_vwv[10-]	*smb_setup*
		smb_bcc	
			smb_param
			smb_data

Table 5-2 Transaction SMB Response Formats

The meaning of the parameters is identical to the definitions above with the parameter names slightly changed; for example, *smb_tprcnt* is the total number of parameter bytes being returned, and is used in the same way as *smb_tpscnt* in the request messages.

As was the case in the request messages, the ordering of *smb_param* and *smb_data* is not required, since *smb_prdisp* and *smb_drdisp* are sufficient to locate each correctly.

5.2.3 Transaction Flow

A small set of rules governs the flow of the various protocol elements making up a transaction, including which request or response type to send at any particular time.

1. The requester sends the first (primary) request which identifies the total bytes (parameters and data) which are to be sent, and contains the setup words, and as many of the parameter and data bytes as fit in a negotiated size buffer. This request also identifies the maximum number of bytes (setup, parameters and data) the responder may return when the transaction is completed. The parameter bytes are immediately followed by the data bytes (the length fields identify the break point). If all the bytes fit in the single buffer, skip to step 4.

2. The responder responds with a single interim response meaning ‘‘OK, send the remainder of the bytes’’, or (if error response) terminate the transaction.
3. The requester then sends a secondary request full of bytes to the responder. This step is repeated until all bytes have been delivered to the responder.
4. The responder sets up and performs the transaction with the information provided.
5. Upon completion of the transaction, if bit 1 of *smb_flag* was not set in the primary request, the responder sends back up to the number of parameter and data bytes requested (or as many as fit in the negotiated buffer size). This step is repeated until all bytes requested have been returned. Fewer than the requested number of bytes (from *smb_mdrcnt* and *smb_mprcnt*) may be returned.

The flow of a transaction when the request parameters and data do *not* all fit in a single buffer is:

requester	–>	TRANSACTION request (data)	>–>	responder
requester	<–<	OK send remaining data	<–	responder
requester	–>	TRANSACTION secondary request 1 (data)	>–>	responder
requester	–>	TRANSACTION secondary request 2 (data)	>–>	responder
requester	–>	TRANSACTION secondary request *n* (data)	>–>	responder
		(responder sets up and performs the TRANSACTION)		
requester	<–<	TRANSACTION response 1 (data)	<–	responder
requester	<–<	TRANSACTION response 2 (data)	<–	responder
requester	<–<	TRANSACTION response *n* (data)	<–	responder

The flow for the Transaction protocol when the request parameters and data do all fit in a single buffer is:

requester	–>	TRANSACTION request (data)	>–>	responder
		(responder sets up and performs the TRANSACTION)		
requester	<–<	TRANSACTION response 1 (data)	<–	responder
		(only one if all data fits in buffer)		
requester	<–<	TRANSACTION response 2 (data)	<–	responder
requester	<–<	TRANSACTION response *n* (data)	<–	responder

Note that the primary request through to the final response make up the complete protocol: thus, the TID, PID, UID and MID are expected to remain constant and can be used by both the responder and requester to route the individual messages of the protocol to the correct process.

The simplest form of a Transaction sends a single primary request and (optionally) receives a single final response. Thus, if the entire Transaction message fits within the size limits for a Datagram (transport-dependent but no smaller than 512 bytes) and reliable delivery of the information is not required, the request and response may be sent/received as NetBIOS datagrams. If the request is sent in a NetBIOS datagram, the reply (if any) must also be a NetBIOS datagram.

5.2.4 SMBtrans Error Code Descriptions

CAE Code	DOS Class	DOS Code	Description
EPERM	ERRDOS	ERRnoaccess	Access denied, the requester's context does not permit the requested function.
	ERRDOS	ERRbadaccess	Invalid open mode.
	ERRSRV	ERRerror	Non-specific error code.
	ERRSRV	ERRinvnid	The tree ID (TID) specified was invalid.
EACCES	ERRSRV	ERRaccess	The requester does not have the necessary access rights within the specified context for the requested function. The context is defined by the TID or the UID.
	ERRSRV	ERRmoredata	There is more data to be returned.

Table 5-3 SMBtrans Error Codes

5.2.5 SMBtrans Deviations

Support for Class 1 mailslots is not mandatory.

5.2.6 Conventions

None.

5.3 Mailslot Usage of SMBtrans

The identifier **\MAILSLOT\name** denotes a mailslot transaction, where the **name** is the mailslot name to apply the transaction against.

Mailslots using unreliable Class 2 mode may be transmitted via NetBIOS datagrams. However, mailslots using reliable Class 1 mode must be transmitted on an established LMX session (reliable delivery is needed).

When Class 1 mailslot transactions are transmitted via an LMX session, a response may still be desired to ensure that the mailslot transaction was delivered to the mailslot without error. Thus the response bit may be zero in *smb_flags* to indicate that the error code associated with the delivery should be returned.

No parameter bytes are sent, and no data bytes or setup words are expected to be returned.

5.3.1 Mailslot Request Parameters

Refer to Section 5.2.1 on page 104 for the structure of requests. Only the parameter names and their contents are described below. The values of unmentioned parameters are determined according to the definition of the request messages.

smb_wct	Must be 17.
smb_tdscnt	This is the total size of data to write to a mailslot (if any).
smb_mprcnt	If bit 1 of *smb_flags* is 0, this must be 2, as a one-word return code is expected.
smb_flags	All flags should be set appropriately. Class 2 mailslots are usually sent with bit 1 set.
smb_timeout	Undefined.
smb_suwcnt	Must be 3.
smb_setup[0]	Interpreted as an opcode. The only defined value is 1=Write Mailslot.
smb_setup[1]	The priority of the message being written to the mailslot. The larger the value the higher the priority.
smb_setup[2]	The class of the message. Class 1 messages are delivered reliably and must be delivered over an existing LMX session. Class 2 messages are not reliable and may be sent over NetBIOS datagrams.
smb_name	This is the mailslot name the message is to be written to. It is an ASCIIZ string of the form **\MAILSLOT\name**.
smb_data	The data to be written to the mailslot.

5.3.2 Mailslot Response Parameters

Refer to Section 5.2.2 on page 106 for the structure of responses. Only the parameter names and their contents are described below. The values of unmentioned parameters are determined according to the definition of the response messages. If a response is generated it should fit in a single response message.

smb_wct	Must be 10.
smb_tprcnt	Must be 2; a single word return code is returned as a parameter.
smb_prcnt	Must be 2.

smb_suwcnt Must be 0.

smb_param This is interpreted as a single word integer containing the mailslot delivery return code. A value of 0 indicates successful delivery.

5.3.3 Special Forms of Mailslot Usage

Servers providing the extended dialect may periodically send a special form of mailslot message, called an Announcement Message, to inform client nodes that the server exists and is ready to accept LMX session connection requests. This message is sent as a Class 2 mailslot message in a NetBIOS datagram sent to an installation-specific NetBIOS group name. No response is permitted.

Clients using the extended dialect may send an Announcement Request Message to request that server nodes available identify themselves via the Announcement Transaction NetBIOS datagram. This message is sent as a Class 2 mailslot message in a NetBIOS datagram sent to an installation-specific NetBIOS group name. No response is permitted.

The Announce and Announce Request messages are sent to a distinguished mailslot named **\MAILSLOT\LANMAN**. The default NetBIOS group name to which the broadcast messages are sent is the 0x20 padded name **LANGROUP**.

Also note that there is no security involved with these protocols. The *smb_tid* and *smb_uid* fields are set to –1 and are ignored by the node receiving this transaction. Each node may apply its own security mechanisms to determine whether to reply to (or send) these protocols.

Announce Mailslot Request Format

Since this is a form of mailslot write transaction, the definitions in the previous section apply; only where values are different or more specific are they specified here. No response is permitted from the responder, so all return counts should be zero. No parameters are sent.

smb_flags Since these messages are sent over a NetBIOS datagram, bit 0 is ignored but should be 0. Bit 1 must be set to indicate no response is to be generated.

smb_setup[2] Must be 2 indicating this is a Class 2 message.

smb_name Must be **\MAILSLOT\LANMAN**.

smb_data This data structure determines whether this is an Announce or Announce Request message.

Announce fields:

Position	Field Name	Description
00	*op_code*	A single word whose value must be 1=Announce.
02	*services*	A double word containing flags indicating, if set, that the Announcer provides the indicated service: Bit 0 Announcer is an LMX client (workstation). Bit 1 Announcer is an LMX server. Bits 2-31 Reserved.
06	*vers_major*	A single byte giving the major version number of the software running on the server.
07	*vers_minor*	A single byte giving the minor version number of the software running on the server.
08	*periodicity*	A single word indicating the number of seconds between announcements generated by the sending system.
10	*node_name*	A null-terminated ASCIIZ string containing the **computername** of the sending node.
after *node_name*	*comment*	A null-terminated ASCIIZ string containing any arbitrary information the sending system wishes to announce.

Announce Request fields:

Position	Field Name	Description
00	*op_code*	A single word whose value must be 2=Announce Request.
02	*node_name*	A null-terminated ASCIIZ string containing the **computername** of the sending node.

5.4 Named Pipe Usage of SMBtrans

Named pipes are an IPC mechanism for communication between a client and server process which resides partially in the file system made available to the client by the server. Many file-oriented operations supported by the SMB protocol are supported on named pipes as well, and have obvious semantics.

Certain operations on named pipes do not easily map onto the file-oriented SMB requests, though; these operations are provided by use of *SMBtrans*. They fall into two broad classes:

- Complex or Unique Named Pipe Operations

  Complex operations that require multiple SMB protocol interchanges can be supported by a single *SMBtrans*. Examples are *CallNmPipe* and *TransactNmPipe*. The *CallNmPipe* operation, provided via *SMBtrans*, permits the *Open*, *Write*, *Read* and *Close* operations to be performed in one transaction on a named pipe. Similarly, the *TransactNmPipe* operation permits the *Write* and *Read* operations to be performed in one transaction.

  RawWriteNmPipe and *RawReadNmPipe* are unique operations that require *SMBtrans* to implement. They cannot be supported via the other SMB protocols.

- Status Operations

  Operations to query and change the modes and attributes of named pipes are required, as are mechanisms to query the status of a named pipe. They are provided via *SMBtrans* as well.

  Named pipes require reliable delivery; thus, the *SMBtrans* transaction must take place on an established LMX session.

  When discussing named pipe requests, the terms *client* and *requester* are used interchangeably, as are *server* and *responder*.

5.4.1 Named Pipe Requests - Detailed Discussion

The identifier **\PIPE\name** denotes a named pipe transaction, where the **name** is the pipe name to apply the transaction against. Note that the named pipe transaction name **\PIPE\LANMAN** is reserved for use by LMX.

The parameters are used as defined in the preceding section on *SMBtrans*. Only parameters whose meaning is more specific for named pipes usage are described here.

smb_wct Must be 16.

smb_msrcnt Must be zero. No named pipe requests cause a response with setup words.

smb_suwcnt Must be 2.

smb_setup[0] This is the specific function to be performed as a result of the request. The values are (in numeric order):

0x01	*SetNmPHandState*	set pipe handle modes.
0x11	*RawReadNmPipe*	read pipe in *raw* (non-message mode).
0x21	*QNmPHandState*	query pipe handle modes.
0x22	*QNmPipeInfo*	query pipe attributes.
0x23	*PeekNmPipe*	read but do not remove data.
0x26	*TransactNmPipe*	write/read operation on pipe.

0x31	*RawWriteNmPipe*	write pipe *raw* (non-message mode).
0x53	*WaitNmPipe*	wait for pipe to be non-busy.
0x54	*CallNmPipe*	open/write/read/close pipe.

smb_setup[1] Either the FID of the pipe on which the function is to be performed, or the priority to be used for the operation; the particular meaning used is defined below with the description of each function.

smb_name An ASCIIZ string of the form **\PIPE\name** specifying the pipe on which the operation is to be performed.

smb_param Specific to the particular operation.

smb_data Specific to the particular operation.

Each of the defined named pipe functions is specified in the following sections in alphabetical order.

5.4.2 CallNmPipe - Function 0x54

This transaction has the combined effect on a named pipe of *SMBopen*, followed by *TransactNmPipe*, followed by *SMBclose*.

This form of transaction neither sends nor receives parameter bytes. The bytes to be written to the pipe are sent in the *smb_data* field of the request message(s), and the bytes read from the pipe are returned in the *smb_data* field of the response(s).

The largest amount of data that can be written to or read from a named pipe is 65536 bytes.

If the timeout specified in *smb_timeout* expires before *smb_mdrcnt* bytes of data can be read from the pipe, the responder should return all the data that could be read.

Message Parameter Values

smb_tdscnt The number of data bytes to be written to the named pipe.

smb_mdrcnt The maximum number of data bytes that may be read from the named pipe and returned in this transaction.

smb_setup[1] Interpreted as a priority (0 is default priority, 1023 is the highest priority). This value is used by the server in determining which request to service when the named pipe server process becomes available.

smb_tdrcnt The total number of bytes that were read from the pipe on the responder and is returned to the requester. This is smaller than *smb_mdrcnt* if a timeout occurred.

5.4.3 PeekNmPipe - Function 0x23

This named pipe transaction is used to read a pipe without removing the read data from the pipe.

No parameters or data are sent by the requester, and no setup words are returned by the responder.

Message Parameter Values

smb_mdrcnt The maximum number of bytes the client wants to peek from the pipe.

smb_setup[1] This should be the FID (handle) of the named pipe on which the peek should be performed.

smb_prcnt Must be 6 (see *smb_param*).

smb_tdrcnt The number of bytes actually read from the pipe. May be smaller than *smb_mdrcnt* if a timeout occurred.

smb_param In the response, this is formatted as follows:

pipecnt An unsigned word giving the number of bytes remaining in the pipe.

msgcnt An unsigned word giving the number of bytes remaining in current message.

pipestat A word indicating the status of the pipe. Possible values are:

— NP_DISCONNECTED (disconnected by server).

— NP_LISTENING (N/A not returned on client end of pipe).

— NP_CONNECTED (connection to server OK).

— NP_CLOSING (server end of pipe closed).

5.4.4 QNmPHandState - Function 0x21

This named pipe transaction returns pipe-specific state information. The values returned are those originally established at the time of opening the pipe or by a subsequent *SetNmPHandState*.

Data is neither sent nor returned, and parameters are not sent. No setup words are returned.

Message Parameter Usage

smb_setup[1] This is the FID (handle) of the named pipe for which information should be returned.

smb_tprcnt Must be 2.

smb_param A two-byte flag word defined as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B	E	0	0	T	T	R	R	——			Icount			——	

where:

B - Blocking	0	reads/writes block if no data available.
	1	reads/writes return immediately if no data.
E - Endpoint	0	client end of pipe.
	1	server end of pipe.
TT - Type of pipe	00	pipe is a byte stream pipe.
	01	pipe is a message pipe.
RR - Read Mode	00	Read pipe as a byte stream.

	01	Read messages from pipe.
Icount		8-bit count to control pipe instancing (N/A).

A server must set the E bit to 0 when responding to this SMB because the handle specified is the client end of the pipe.

5.4.5 QNmPipeInfo - Function 0x22

This named pipe transaction returns information about a pipe.

No data is sent by the requester, and no parameters or setup words are returned by the responder.

Message Parameter Values

smb_mdrcnt The maximum amount of information data the requester would like returned.

smb_setup[1] The FID (handle) of the pipe for which information should be returned.

smb_param A signed integer indicating the level of information requested. Only level 1 is currently defined.

smb_tdrcnt The length of the data being returned. This may be less than *smb_mdrcnt*.

smb_data The pipe information being returned. The format of the data depends on the information level requested (in *smb_param*). The information may be truncated if *smb_mdrcnt* is smaller than the number of bytes needed for the information. For level 1, the data contains the following pieces of information, in this order with no padding:

- An unsigned word containing the actual size of buffer for outgoing (server) I/O.
- An unsigned word containing the actual size of buffer for incoming (client) I/O.
- An unsigned byte indicating the maximum allowed number of instances of this pipe.
- An unsigned byte indicating the current number of instances for this pipe.
- An unsigned byte giving the length of pipe name, including the terminating null.
- An ASCIIZ string containing the name of the pipe, in the form **\PIPE\name**.

5.4.6 RawReadNmPipe - Function 0x11

This named pipe transaction reads a named pipe without removing record information. Bytes are read directly from a pipe, regardless of whether it is a message or byte pipe. For a byte stream pipe, this transaction behaves exactly like *SMBread*. For a message pipe, this is exactly like reading the pipe in byte stream read mode, except message headers can also be returned in the buffer (note that message headers are always returned completely - never split at a byte boundary).

No parameters are sent or received, no data is sent, and no setup words are received.

Message Parameter Values

smb_mdrcnt The number of bytes to be read from the pipe.

*smb_setup[*1] The FID (handle) of the named pipe to be read from.

smb_tdrcnt The number of bytes actually read.

smb_data The bytes read from the named pipe. Where more than one response is required subsequent responses would indicate the data bytes remaining.

5.4.7 RawWriteNmPipe - Function 0x31

This named pipe transaction writes a named pipe without adding record information. It causes bytes to be put directly into a pipe, regardless of whether it is a message or byte pipe. The data includes message headers if it is a message pipe. This call ignores the blocking/non-blocking state and always acts in a blocking manner. It returns only after all bytes have been written.

No parameters are sent in the request, and no data is returned in the response; also, no setup words are returned.

Message Parameter Values

smb_mprcnt Must be 2.

smb_setup[1] This should contain the FID (handle) of the named pipe to which the bytes should be written.

smb_data The data to be written.

smb_param In the response, this is an unsigned integer indicating the number of bytes actually written to the pipe.

5.4.8 SetNmPHandState - Function 0x01

This named pipe transaction sets pipe-specific handle states. Only the read mode (byte versus message) and blocking/non-blocking mode of a named pipe can be changed. Some combinations of parameters may be illegal and rejected as an error.

No data is sent or returned, and no parameters or setup words are returned.

Message Parameter Values

smb_setup[1] The FID (handle) of the pipe whose state should be changed.

smb_param A two-byte bit field defined as follows:

1 5	1 4	1 3	1 2	1 1	1 0	9	8	7	6	5	4	3	2	1	0
B	0	0	0	0	0	R	R	0	0	0	0	0	0	0	0

where:

B - Blocking	0	reads/writes block if no data available.
	1	reads/writes return immediately if no data.
RR - Read Mode	00	Read pipe as a byte stream.
	01	Read messages from pipe.

5.4.9 TransactNmPipe - Function 0x26

This named pipe transaction combines a write and read operation on a named pipe. It provides an optimum way to implement transaction-oriented, or request-reply, dialogues.

TransactNmPipe fails if the pipe currently contains any unread data or is not in message read mode. Otherwise the call writes the entire request data bytes to the pipe and then reads a response from the pipe and returns it in the data bytes area of the response protocol.

The state of blocking/non-blocking has no effect on this protocol; that is, the transaction does not return until a message has been read from the pipe.

No parameters are sent or returned, and no setup words are returned.

Message Parameter Values

smb_mdrcnt	The maximum amount of data that may be returned in this transaction. If the response read from the pipe is larger than this value, [ERRmoredata] is returned.
smb_setup[1]	The FID (handle) of the named pipe on which the write and read should be performed.
smb_data	In the request, the data to be written; in the response, the data that was read.

5.4.10 WaitNmPipe - Function 0x53

This form of named pipe transaction waits for the availability of a named pipe instance.

DosWaitNmPipe allows an application to wait for a pipe when all available instances are currently busy. This might be used when the error [ERRpipebusy] is returned from an attempt to open a named pipe.

The server waits up to *smb_timeout* milliseconds for a pipe of the name given to become available. Although the timeout is specified in milliseconds, by the time the timeout occurs and the client receives the response indicating this fact, much more time than specified may have occurred.

Both the request and response contain no data and parameter bytes. If no error is returned, the pipe instance *may* be available. This request does *not* reserve the pipe, thus all waiting programmes may race to get the pipe now available. The losers again get [ERRpipebusy] on open attempts.

The *smb_setup*[1] field is interpreted as a priority. The priority values range from 0 (use server default) to 1023 (highest priority). The server may use the priority in determining which process to notify when a pipe becomes available.

Chapter 6

SMB Protocols for Messaging

This chapter supersedes information given in Appendix C of the referenced document **Protocols for X/Open PC Interworking: SMB**. Many terms are used within this section which are defined in **Protocols for X/Open PC Interworking: SMB**.

6.1 Introduction

These commands provide a message delivery system between users of systems participating in the network. The message commands cannot use LMX sessions established for the file-sharing commands. A separate NetBIOS session, dedicated to messaging, must be established.

Messaging services should support message forwarding. By convention NetBIOS names used for message delivery have a suffix (in byte 16) of 0x03; forwarded names have a suffix of 0x05. The algorithm for sending messages is first to attempt to deliver the message to the forwarded name, and only if this fails to attempt to deliver to the normal name. In all of the interfaces where the NetBIOS name is to be supplied, the maximum number of bytes that can be supplied for the name is 15. For additional information, see Appendix D of the referenced document **Revised XTI (X/Open Transport Interface)** and Section 3.8 of **Protocols for X/Open PC Interworking: SMB**.

The following sections make reference to *typed* buffers. Two buffer types are used within the protocols described in this chapter: type 01 and type 04. For typed buffers, the first byte of the buffer contains the type indicator (0x01 for type 01 or 0x04 for type 04). A type 01 buffer is a buffer block where the first word after the type indicator contains the size of the data contained in the buffer. A type 04 buffer indicates that the following data is an ASCIIZ string. For additional information on typed buffers see Section 4.2.5 of **Protocols for X/Open PC Interworking: SMB**.

6.2 SMBsends Specification

6.2.1 SMBsends Detailed Description

This core request sends a short message (up to 128 bytes in length) to a single destination system. This SMB may be sent from any system, client or server.

6.2.2 SMBsends Deviations

None.

6.2.3 SMBsends Field Descriptions

From Sender		To Sender	
Field Name	**Field Value**	**Field Name**	**Field Value**
smb_com	*SMBsends*	*smb_com*	*SMBsends*
smb_wct	0	*smb_wct*	0
smb_bcc	min=7	*smb_bcc*	0
smb_buf[ ]	*smb_orig*		
	smb_dest		
	smb_message		

From Sender Description

smb_orig A type 04 buffer containing the name of the sender (maximum length is 15 bytes).

smb_dest A type 04 buffer containing the name of the destination (maximum length is 15 bytes).

smb_message A type 01 data block containing the message. The length of the data in the block is 128 bytes or less.

6.2.4 SMBsends Error Code Descriptions

The following error codes indicate that the message was received with the appropriate conditions:

DOS Class	DOS Code	Description
SUCCESS	SUCCESS	The request was successful.
SUCCESS	BUFFERED	The message has been buffered.
SUCCESS	LOGGED	The message has been logged.
SUCCESS	DISPLAYED	The message has been displayed.

Table 6-1 SMBsends Success Codes

The following error codes indicate a failure in the message transmission:

CAE Code	DOS Class	DOS Code	Description
-	ERRSRV	ERRerror	Non-specific error code.
-	ERRSRV	ERRpaused	Server is paused.
-	ERRSRV	ERRmsgoff	Not receiving messages.
ENOMEM	ERRSRV	ERRnoroom	No buffer space was available on the server; the message was dropped.

Table 6-2 SMBsends Error Codes

6.2.5 SMBsends Preconditions

An LMX session is established.

6.2.6 SMBsends Postconditions

None.

6.2.7 Conventions

None.

6.3 SMBsendb Specification

6.3.1 SMBsendb Detailed Description

This Core SMB request sends a short message (up to 128 bytes in length) to every system in the network. The broadcast mechanism is the responsibility of the transport layer and is outside the scope of this document. Refer to Chapters 14 and 15 of **Protocols for X/Open PC Interworking: SMB**.

6.3.2 SMBsendb Deviations

None.

6.3.3 SMBsendb Field Descriptions

From Sender		To Sender
Field Name	**Field Value**	
smb_com	*SMBsendb*	(No response)
smb_wct	0	
smb_bcc	min = 8	
smb_buf[ ]	*smb_orig*	
	smb_dest	
	smb_message	

From Sender Description

smb_orig A type 04 buffer containing the name of the sending system (maximum length is 15 bytes).

smb_dest A type 04 buffer giving the destination name. This is always a string containing only an asterisk string ("*").

smb_message A type 01 data block containing the message. This block is 128 bytes or less in length.

6.3.4 SMBsendb Error Code Descriptions

Since no response is generated to this request, and since the broadcast service is not intended to be reliable, no errors can be returned to the sender.

6.3.5 SMBsendb Preconditions

An LMX session is established.

6.3.6 SMBsendb Postconditions

None.

6.3.7 Conventions

None.

6.4 SMBsendstrt Specification

6.4.1 SMBsendstrt Detailed Description

This core requests the beginning of a multi-part message. The total length of all segments in a multi-block message must be no greater than 1600 bytes.

6.4.2 SMBsendstrt Deviations

None.

6.4.3 SMBsendstrt Field Descriptions

From Sender		To Sender	
Field Name	**Field Value**	**Field Name**	**Field Value**
smb_com	*SMBsendstrt*	*smb_com*	*SMBsendstrt*
smb_wct	0	*smb_wct*	1
smb_bcc	min = 0	*smb_vwv*[0]	*smb_grpid*
smb_buf[ ]	*smb_orig*	*smb_bcc*	0
	smb_dest		

From Sender Description

smb_orig A type 04 buffer containing the name of the sender (maximum length is 15 bytes).

smb_dest A type 04 buffer containing the name of the destination (maximum length is 15 bytes). The same constraint as for *smb_orig* applies.

To Sender Description

smb_grpid This is an unsigned short which contains the message group ID. The sender places this value in all other SMB requests which form part of the multi-block message started with this request.

6.4.4 SMBsendstrt Error Code Descriptions

The following error codes indicate that the message was received with the appropriate conditions:

DOS Class	DOS Code	Description
SUCCESS	SUCCESS	The request was successful.
SUCCESS	BUFFERED	The message has been buffered.
SUCCESS	LOGGED	The message has been logged.
SUCCESS	DISPLAYED	The message has been displayed.

Table 6-3 SMBsendstrt Success Codes

The following error codes indicate a failure in the message transmission:

CAE Code	DOS Class	DOS Code	Description
-	ERRSRV	ERRerror	Non-specific error code.
-	ERRSRV	ERRpaused	Server is paused.
-	ERRSRV	ERRmsgoff	Not receiving messages.
ENOMEM	ERRSRV	ERRnoroom	No buffer space was available on the server; the message was dropped.

Table 6-4 SMBsendstrt Error Codes

6.4.5 SMBsendstrt Preconditions

An LMX session is established.

6.4.6 SMBsendstrt Postconditions

None.

6.4.7 Conventions

None.

6.5 SMBsendtxt Specification

6.5.1 SMBsendtxt Detailed Description

This core request is used to send a segment of a multi-block message. The message must have been started by an SMBsendstrt request on the same LMX session with the same message group ID.

Each segment of a multi-block message may be no longer than 128 bytes.

6.5.2 SMBsendtxt Deviations

None.

6.5.3 SMBsendtxt Field Descriptions

From Sender		To Sender	
Field Name	**Field Value**	**Field Name**	**Field Value**
smb_com	*SMBsendtxt*	*smb_com*	*SMBsendtxt*
smb_wct	1	*smb_wct*	0
smb_vwv[0]	*smb_grpid*	*smb_bcc*	0
smb_bcc	min=3		
smb_buf[ ]	*smb_message*		

From Sender Description

smb_grpid This unsigned short is the message group ID, as returned from the receiver in its response to the SMBsendstrt request. This is used to identify the multi-block message of which this segment is a part.

smb_message A type 01 data block containing the text of this segment of the multi-block message.

6.5.4 SMBsendtxt Error Code Descriptions

The following error codes indicate that the message was received with the appropriate conditions:

DOS Class	DOS Code	Description
SUCCESS	SUCCESS	The request was successful.
SUCCESS	BUFFERED	The message has been buffered.
SUCCESS	LOGGED	The message has been logged.
SUCCESS	DISPLAYED	The message has been displayed.

Table 6-5 SMBsendtxt Success Codes

The following error codes indicate a failure in the message transmission:

CAE Code	DOS Class	DOS Code	Description
-	ERRSRV	ERRerror	Non-specific error code.
-	ERRSRV	ERRinvnid	The Tree ID (TID) is not valid for this request.
-	ERRSRV	ERRpaused	Server is paused.
-	ERRSRV	ERRmsgoff	Not receiving messages.
ENOMEM	ERRSRV	ERRnoroom	No buffer space was available on the server; the message was dropped.

Table 6-6 SMBsendtxt Error Codes

6.5.5 SMBsendtxt Preconditions

An LMX session is established.

6.5.6 SMBsendtxt Postconditions

None.

6.5.7 Conventions

None.

6.6 SMBsendend Specification

6.6.1 SMBsendend Detailed Description

This core request is used to indicate the end of a multi-block message. It is sent after all segments of the message have been received and indicates the message is complete.

6.6.2 SMBsendend Deviations

None.

6.6.3 SMBsendend Field Descriptions

From Sender		To Sender	
Field Name	**Field Value**	**Field Name**	**Field Value**
smb_com	*SMBsendend*	*smb_com*	*SMBsendend*
smb_wct	1	*smb_wct*	0
smb_vwv[0]	*smb_grpid*	*smb_bcc*	0
smb_bcc	0		

From Sender Description

smb_grpid This unsigned short is the message group ID, as returned from the receiver in its response to the *SMBsendstrt* request. This is used to identify the multi-block message of which this segment is a part.

6.6.4 SMBsendend Error Code Descriptions

CAE Code	DOS Class	DOS Code	Description
	ERRSRV	ERRerror	Non-specific error code.
	ERRSRV	ERRpaused	Server is paused.
	ERRSRV	ERRmsgoff	Not receiving messages.

Table 6-7 SMBsendend Error Codes

6.6.5 SMBsendend Preconditions

An LMX session is established.

6.6.6 SMBsendend Postconditions

None.

6.6.7 Conventions

None.

6.7 SMBsendfwd Specification

6.7.1 SMBsendfwd Detailed Description

This core request is sent to a system to instruct it to accept messages sent to a forwarded name. This request is sent on the same LMX session as other core SMB requests.

6.7.2 SMBsendfwd Field Descriptions

From Sender		To Sender	
Field Name	**Field Value**	**Field Name**	**Field Value**
smb_com	*SMBfwdname*	*smb_com*	*SMBfwdname*
smb_wct	0	*smb_wct*	0
smb_bcc	min=2	*smb_bcc*	0
smb_buf[ ]	*smb_fwdname*		

From Sender Description

smb_fwdname A type 04 buffer containing the forwarded name on which the receiver should be prepared to receive messages (maximum length is 15 bytes).

6.7.3 SMBsendfwd Error Code Descriptions

CAE Code	DOS Class	DOS Code	Description
	ERRSRV	ERRerror	Non-specific error code.
	ERRSRV	ERRrmuns	Too many remote user names.

Table 6-8 SMBsendfwd Error Codes

6.7.4 SMBsendfwd Preconditions

An LMX session is established.

6.7.5 SMBsendfwd Postconditions

None.

6.7.6 Conventions

None.

6.8 SMBcancelf Specification

6.8.1 SMBcancelf Detailed Description

This core request cancels the effect of a prior *SMBsendfwd* request. The receiving system no longer accepts messages for the designated system name. This request is sent on the same LMX session as other core SMB requests.

6.8.2 SMBcancelf Deviations

None.

6.8.3 SMBcancelf Field Descriptions

From Sender		To Sender	
Field Name	**Field Value**	**Field Name**	**Field Value**
smb_com	*SMBcancelf*	*smb_com*	*SMBcancelf*
smb_wct	0	*smb_wct*	0
smb_bcc	min=2	*smb_bcc*	0
smb_buf[ ]	*smb_fwdname*		

From Sender Description

smb_fwdname A type 04 buffer containing the forwarded name on which the receiver should be prepared to receive messages (maximum length is 15 bytes).

6.8.4 SMBcancelf Error Code Descriptions

CAE Code	DOS Class	DOS Code	Description
	ERRSRV	ERRerror	Non-specific error code.
	ERRSRV	ERRinvnid	The Tree ID is not valid for this request.

Table 6-9 SMBcancelf Error Codes

6.8.5 SMBcancelf Preconditions

An LMX session is established.

6.8.6 SMBcancelf Postconditions

None.

6.8.7 Conventions

None.

6.9 SMBgetmac Specification

6.9.1 SMBgetmac Detailed Description

This core request returns the messaging (system) name of the receiving system. Its most common usage is with forwarded names; a system could send an *SMBgetmac* request to a forwarded name to find the identity of the system to which that name had been forwarded. This name could then be used, for example, to cancel the forwarding of the original name (via *SMBcancelf*).

6.9.2 SMBgetmac Deviations

None.

6.9.3 SMBgetmac Field Descriptions

From Sender		To Sender	
Field Name	**Field Value**	**Field Name**	**Field Value**
smb_com	*SMBgetmac*	*smb_com*	*SMBgetmac*
smb_wct	0	*smb_wct*	0
smb_bcc	0	*smb_bcc*	min=2
		smb_buf[ ]	*smb_name*

To Sender Description

smb_name A type 04 buffer containing the actual messaging (system) name of the receiving system (maximum length is 15 bytes).

6.9.4 SMBgetmac Error Code Descriptions

CAE Code	DOS Class	DOS Code	Description
	ERRSRV	ERRerror	Non-specific error code.
	ERRSRV	ERRinvnid	The Tree ID in the request is invalid.

Table 6-10 SMBgetmac Error Codes

6.9.5 SMBgetmac Preconditions

An LMX session is established.

6.9.6 SMBgetmac Postconditions

None.

6.9.7 Conventions

None.

Glossary

8.3 format name
A name consisting of two components. A one to eight character name must be present and an optional one to three character extension may be added. The name is separated from the extension by a period (.).

ACL
(Access Control List) A list used to control access to a file or resource. The list contains the user IDs and/or group IDs that are allowed access to the file or resource.

API
(Application Programming Interface) Function definitions for programmers.

ASCIIZ
A zero-terminated ASCII string.

broadcast
The function of delivering a given packet to all hosts that are attached to the broadcasting delivery system. Broadcasting is implemented both at the hardware and the software levels.

byte
8 bits.

CAE
Common Applications Environment.

chaining
Transmission of more than one SMB request in a single transport PDU.

client-server
The distributed system model where a requesting program (the client) interacts with a program that can satisfy the request (the server). The client initiates the interaction and may wait for the server to respond.

computernames
A name that refers to a system that may or may not be running services. These names have a length restriction of 15 characters and map to NetBIOS names.

connection-oriented service
A service provided between two applications along which data is passed in a sequenced and reliable way.

data encapsulation
The way a lower-level protocol accepts a message from a higher-level protocol and places it in the data portion of the low-level frame.

daemon
A process that is not associated with any user. This sort of process performs system-wide functions, for example, administration, control of networks and execution-dependent activities.

domain name
A name used by the messaging IPC mechanism to group users within the network.

DWORD
Consists of four bytes, ordered such that the low-order WORD precedes the high WORD. See also WORD.

extended dialect
An LMX dialect providing extra services. See **Protocols for X/Open PC Interworking: SMB**.

FID
(File ID) A unique number associated with a file to enable it to be identified.

FIFO
(First In First Out) One of the file types supported on an XSI system. A FIFO, the alternative name for a pipe, differs from a regular file because its data is transient; that is, once data is read from the pipe it cannot be read again.

fork
The XSI system call which is used to create a new process. The process created is a duplicate of the calling process.

GID
(Group ID) A unique number associated with a group to enable it to be identified.

ID
Identification.

interoperability
The ability of software and hardware on multiple machines and from multiple vendors to communicate effectively.

IPC
(Interprocess Communication) Methods by which two or more processes can communicate, for example, formatted data streams or shared memory.

IPC names
This is the collection of names for named pipes, mailslots and messaging. These names must be consistent with the UNC format. That is, any 8.3 format name using characters that are legal within the confines of the receiving system's file system are acceptable. These names are case-insensitive. The name can imply the existence of a directory. For example, the mailslot name **\MAILSLOT\GSP\SERVICE** can be used to put the mailslot SERVICE within the ''directory'' GSP. It should be noted, however, that no actual directory exists. Therefore it is not necessary to create the ''directory'' name prior to the creation of the mailslot name. Each implementation of LMX has some restriction on the length of an IPC name. All LMX implementations guarantee that IPC names of at least 128 characters are supported.

IPC$
The share name for LMX IPC resource that allows a client system to connect to named pipes and mailslots.

LAN
(Local Area Network) A physical network that operates at a high speed over short distances, for example, Ethernet.

LMX server
The implementation of the specification **Protocols for X/Open PC Interworking: SMB** on a CAE system.

LMX server name
The name used by users of LMX servers to identify the specific LMX server desired.

LMX session
A logical communication connection between the LMX server and a client system where LMX IPC or LMX data sharing can take place. Reference **Protocols for X/Open PC Interworking: SMB**.

mailslot
A unidirectional form of communication between cooperating processes on a network.

mailslot class
An indication of the expected service of a mailslot. Class 1 is guaranteed delivery. Class 2 is not guaranteed delivery.

MBZ
(Must Be Zero) Reserved fields are often defined MBZ.

message group ID
A unique number associated with a group of messages to enable it to be identified.

messenger
A service that allows the messaging of LMX IPC to be received on a system.

messaging
A form of IPC between network systems that does not guarantee delivery of the message.

MID
(Multiplex Identifier) A number which uniquely identifies a protocol request and response within a process.

multicast
A method by which copies of a single packet are passed to a selected subset of all destinations. Broadcast is a special case of multicast whereby the subset of destinations receiving a copy of the packet is the entire set of destinations.

named pipe
An interprocess communication mechanism defined by the extended SMB specification, as defined in **Protocols for X/Open PC Interworking: SMB**. Also a FIFO.

NetBIOS
(Network Basic Input Output System) The *de facto* standard programmatic interface to networks for DOS systems.

NetBIOS broadcast datagram
Similar to a NetBIOS datagram except the destination address indicates that all cooperating systems on the network are to receive and process the packet. See Chapters 14 and 15 of **Protocols for X/Open PC Interworking: SMB**.

NetBIOS datagram
A packet sent independently of the others in the network. It contains the source and destination addresses as well as the data.

NetBIOS group names
Similar to a NetBIOS datagram except the destination address represents a group of systems instead of a single system. See Chapters 14 and 15 of **Protocols for X/Open PC Interworking: SMB**.

NetBIOS names
Names that refer to NetBIOS naming conventions. Not all legal NetBIOS names are necessarily legal server names. See Chapters 14 and 15 of **Protocols for X/Open PC Interworking: SMB**.

NULL
This is a null pointer obtained by converting the number 0 into a pointer, for example, (char *) 0. NULL is defined in the XSI header file <**stdio.h**>.

network
See LAN.

octet
8 bits.

PDU
(Protocol Data Unit) The basic unit of data manipulated by a protocol.

PID
(Process ID) The number assigned to a process so that it can be uniquely identified.

requester
An entity which initiates a transport connection with a responder. See also responder.

resource names
Names that refer to the public name by which a resource is made available on the LMX server for access from client systems (for example, the IPC$ share).

responder
An entity with which an initiator wishes to establish a transport connection.

RFC
(Request for Comments) The name of a series of notes that contain surveys, measurements, ideas, techniques and observations, as well as proposed and accepted Internet protocol standards.

server names
Names that refer to LMX systems that are used to provide services.

SMB
(Server Message Block) A protocol which allows a set of computers to access shared resources as if they were local. The core protocol was developed by Microsoft Corporation and Intel, and the extended protocol was developed by Microsoft Corporation.

share security
A form of LMX security where permissions and protection on data are managed by knowledge of the password to a share name.

TBD
(To Be Defined) Further detail will be provided at a later time.

TID
(Tree connect IDentifier) A TID uniquely identifies a resource connection between an LMX client and an LMX server.

type byte
The first byte of a typed buffer. See **Protocols for X/Open PC Interworking: SMB**.

typed buffer
A buffer where the first byte indicates the type of data which follows. For example, the first

byte could indicate an ASCIIZ string, or a buffer where the size of the data is contained in the WORD immediately following the type byte.

UID

(User ID) A token representing an authenticated <*username, password*> tuple. UIDs are registered by the redirectors.

UNC

(Uniform Naming Convention) Names constructed from names following an 8.3 format and separated by a backslash (\).

user names

Names that refer to individual users of the LMX services.

user security

A form of LMX security where individual users on the network are given a UID and access permissions are based on this UID.

WORD or word

Consists of two bytes, ordered such that the low-order byte precedes the high byte.

Index