Novell
Developer Kit

SINGLE AND INTRA-FILE SERVICES

Novell®

Novell Trademarks

AppNotes is a registered trademark of Novell, Inc.

AppTester is a registered trademark of Novell, Inc. in the United States.

ASM is a trademark of Novell, Inc.

Beagle is a trademark of Novell, Inc.

BorderManager is a registered trademark of Novell, Inc.

BrainShare is a registered service mark of Novell, Inc. in the United States and other countries.

C3PO is a trademark of Novell, Inc.

Certified Novell Engineer is a service mark of Novell, Inc.

Client32 is a trademark of Novell, Inc.

CNE is a registered service mark of Novell, Inc.

ConsoleOne is a registered trademark of Novell, Inc.

Controlled Access Printer is a trademark of Novell, Inc.

Custom 3rd-Party Object is a trademark of Novell, Inc.

DeveloperNet is a registered trademark of Novell, Inc., in the United States and other countries.

DirXML is a registered trademark of Novell, Inc.

eDirectory is a trademark of Novell, Inc.

Excelerator is a trademark of Novell, Inc.

exteNd is a trademark of Novell, Inc.

exteNd Director is a trademark of Novell, Inc.

exteNd Workbench is a trademark of Novell, Inc.

FAN-OUT FAILOVER is a trademark of Novell, Inc.

GroupWise is a registered trademark of Novell, Inc., in the United States and other countries.

Hardware Specific Module is a trademark of Novell, Inc.

Hot Fix is a trademark of Novell, Inc.

Hula is a trademark of Novell, Inc.

iChain is a registered trademark of Novell, Inc.

Internetwork Packet Exchange is a trademark of Novell, Inc.

IPX is a trademark of Novell, Inc.

IPX/SPX is a trademark of Novell, Inc.

jBroker is a trademark of Novell, Inc.

Link Support Layer is a trademark of Novell, Inc.

LSL is a trademark of Novell, Inc.

ManageWise is a registered trademark of Novell, Inc., in the United States and other countries.

Mirrored Server Link is a trademark of Novell, Inc.

Mono is a registered trademark of Novell, Inc.

MSL is a trademark of Novell, Inc.

My World is a registered trademark of Novell, Inc., in the United States.

NCP is a trademark of Novell, Inc.

NDPS is a registered trademark of Novell, Inc.

NDS is a registered trademark of Novell, Inc., in the United States and other countries.

NDS Manager is a trademark of Novell, Inc.

NE2000 is a trademark of Novell, Inc.

NetMail is a registered trademark of Novell, Inc.

NetWare is a registered trademark of Novell, Inc., in the United States and other countries.

NetWare/IP is a trademark of Novell, Inc.

TTS is a trademark of Novell, Inc.

Universal Component System is a registered trademark of Novell, Inc.

Virtual Loadable Module is a trademark of Novell, Inc.

VLM is a trademark of Novell, Inc.

Yes Certified is a trademark of Novell, Inc.

ZENworks is a registered trademark of Novell, Inc., in the United States and other countries.

Third-Party Materials

All third-party trademarks are the property of their respective owners.

Contents

About This Guide

This guide contains information primarily for services with functions that operate on one file at a time, or that perform operations within a single file. It describes the following:

- Chapter 3, "AFP Functions," on page 23
- Chapter 7, "Direct File System Functions," on page 85
- Chapter 10, "DOS Partition Functions," on page 123
- Chapter 14, "Extended Attribute Functions," on page 153
- Chapter 17, "Operating System I/O Functions," on page 201
- Chapter 19, "Stream I/O Functions," on page 251
- Chapter 22, "Synchronization Functions," on page 339
- Chapter 25, "Server-Based Extended Attribute Functions," on page 425
- Chapter 28, "Server-Based Synchronization Functions," on page 447

For functionality that operates principally on multiple files, refer to NDK: Multiple and Inter-File Services.

Feedback

We want to hear your comments and suggestions about this manual and the other documentation included with this product. Please use the User Comments feature at the bottom of each page of the online documentation.

Documentation Updates

For the most recent version of this guide, see NLM and NetWare Libraries for C (including CLIB and XPlat) (http://developer.novell.com/ndk/clib.htm)

Additional Information

For information about other CLib and XPlat interfaces, see the following guides:

- *NDK: NLM Development Concepts, Tools, and Functions*
- *NDK: Program Management*
- *NDK: NLM Threads Management*
- *NDK: Connection, Message, and NCP Extensions*
- *NDK: Multiple and Inter-File Services*
- *NDK: Internationalization*
- *NDK: Volume Management*
- *NDK: Client Management*
- *NDK: Network Management*
- *NDK: Server Management*
- *NDK: Unicode*

- *NDK: Sample Code*
- *NDK: Getting Started with NetWare Cross-Platform Libraries for C*
- *NDK: Bindery Management*

For CLib source code projects, visit Forge (http://forge.novell.com).

For help with CLib and XPlat problems or questions, visit the NLM and NetWare Libraries for C (including CLIB and XPlat) Developer Support Forum (http://developer.novell.com/ndk/devforums.htm). There are two for NLM development (XPlat and CLib) and one for Windows XPlat development.

Documentation Conventions

In this documentation, a greater-than symbol (>) is used to separate actions within a step and items within a cross-reference path.

A trademark symbol (®, ™, etc.) denotes a Novell trademark. An asterisk (*) denotes a third-party trademark.

AFP Concepts

This documentation describes AFP, its functions, and features.

1.1 File Name and Path Conventions

AFP directories and files differ from their counterparts in NetWare® in the length of file names and the way naming paths are designated. AFP names can contain from 1 to 31 characters comprised of any ASCII character between 1 and 255 except the colon or the NULL character. A NetWare server automatically generates short names (DOS-style file names) for all AFP directories, as well as for any files created or accessed from DOS. The server maintains both the long name and the short name for each AFP directory and file.

Be aware that although most AFP functions use AFP directory paths, some require a directory path in NetWare format, and that AFP and NetWare formats cannot be mixed for entry.

1.2 Data and Resource Forks

AFP files are divided into a data fork and a resource fork. The data fork stores data formatted according to the creator's discretion. The resource fork, if present, stores data understood in prescribed formats, such as code, icons, menu bars, alerts, version information, and execution behavior.

From the AFP standpoint, a DOS file is a data file with no resource fork or long name. To endow a DOS file with a Macintosh name, the NetWare® OS permits Macintosh users to give the file a name in the Macintosh name space. Otherwise, Macintosh users see the DOS name just as DOS users do.

1.3 Entry IDs

The AFP entry ID is similar to the NetWare® directory handle. The AFP long file name relates to an AFP entry ID that represents some portion of the file's directory path. However, AFP and NetWare conventions are not interchangeable. Never mix NetWare directory handles with long names or AFP entry IDs with short names.

AFP Services includes the following functions for returning AFP entry IDs using a NetWare directory handle, a long name, or a short name.

1.4 File Information

AFP Services include three functions that access AFP file information:

NWAFPGetFileInformation (page 43) and NWAFPScanFileInformation (page 52) return file information in an NW_AFP_FILE_INFO (page 64) structure. NWAFPSetFileInformation (page 56) uses an NW_AFP_SET_INFO (page 67) structure to modify file information. AFP file information includes the following items:

- The AFP ID of the entry
- The AFP ID of the parent of the entry
- File or directory attributes
- Data fork size
- Resource fork size
- Number of files and subdirectories contained in the entry
- Dates and times the entry was created, accessed, modified, and backed up
- Macintosh Finder information
- AFP long name
- The object ID that created or last modified the entry
- NetWare® name in the DOS or primary name space
- Access privileges of the client
- Apple Pro DOS information (Apple II Information)

All three functions include a request mask parameter that indicates the information to be returned or modified. The request mask defines the following values:

First Byte:

Bit 0 = AFP_GET_ATTRIBUTES
Bit 1 = AFP_GET_PARENT_ID
Bit 2 = AFP_GET_CREATE_DATE
Bit 3 = AFP_GET_ACCESS_DATE
Bit 4 = AFP_GET_MODIFY_DATE/TIME
Bit 5 = AFP_GET_BACKUP_DATE/TIME
Bit 6 = AFP_GET_FINDER_INFO
Bit 7 = AFP_GET_LONG_NAME

Second Byte:

Bit 0 = AFP_GET_ENTRY_ID
Bit 1 = AFP_GET_DATE_FORK_LEN
Bit 2 = AFP_RESOURCE_LEN
Bit 3 = AFP_GET_NUM_OFFSPRING
Bit 4 = AFP_GET_OWNER_ID
Bit 5 = AFP_GET_SHORT_NAME
Bit 6 = AFP_GET_ACCESS_RIGHTS
Bit 7 = undefined

1.5 Mac OS Finder Information

The Mac OS uses Finder information—the file type, the icon's location in its parents window, and assorted file flags—to display files on the desktop. Operations such as creating an AFP file or directory require Finder information. If you pass a NULL value for the Finder information when you create a file, the Mac OS automatically creates Finder information.

AFP Tasks

2

This documentation describes common tasks associated with AFP.

2.1 Checking for AFP Support

NetWare® servers support AFP files on a volume-by-volume basis. However, this support is optional. The appropriate name space NLM™ application must be loaded at the server, and AFP support must be enabled on the volume. This is equivalent to asking if MAC.NAM (or the Macintosh name space NLM) is loaded for the volume in question.

Before attempting to perform AFP operations on an entry, call NWAFPSupported (page 60) to make sure the AFP name space is supported on the NetWare volume.

2.2 Operating on AFP Directory Entries

Always use AFP Services to create, delete, and rename AFP directory entries. The File Access Services used to access other NetWare® files cannot perform these operations since they are unable to preserve the relationship between the files data and resource forks.

AFP Services include the following functions to operate on AFP files:

- NWAFPCreateDirectory (page 27)
- NWAFPCreateFile (page 29)
- NWAFPDelete (page 32)
- NWAFPOpenFileFork (page 46)
- NWAFPRename (page 49)

These functions take a combination AFP entry ID and path string that identifies the entry. Also, the string should be length-preceded, the initial byte indicating the length of the string.

AFP Functions

3

This documentation alphabetically lists the AFP functions and describes their purpose, syntax, parameters, and return values.

NWAFPAllocTemporaryDirHandle

Allocates a directory handle for an AFP directory

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: AFP

Syntax

```
#include <nwafp.h>
or
#include <nwcalls.h>

NWCCODE NWAPI  NWAFPAllocTemporaryDirHandle (
   NWCONN_HANDLE        conn,
   nuint16              volNum,
   nuint32              AFPEntryID,
   constr nstr8 N_FAR  *AFPPath,
   NWDIR_HANDLE N_FAR  *dirHandle,
   pnuint8              accessRights);
```

Delphi Syntax

```
uses calwin32

Function NWAFPAllocTemporaryDirHandle
  (conn : NWCONN_HANDLE;
   volNum : nuint16;
   AFPEntryID : nuint32;
   const AFPPathString : pnstr8;
   Var dirHandle : NWDIR_HANDLE;
   accessRights : pnuint8
) : NWCCODE;
```

Parameters

conn

   (IN) Specifies the NetWare® server connection handle.

volNum

   (IN) Specifies the volume number of the directory entry location.

AFPEntryID

   (IN) Specifies the AFP base ID.

AFPPath

(IN) Points to the AFP style directory path relative to `AFPEntryID`.

dirHandle

(OUT) Points to the NetWare directory handle.

accessRights

(OUT) Points to the effective rights the requesting user has on the directory.

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x8805	NET_RECV_ERROR
0x890A	NLM_INVALID_CONNECTION
0x8988	INVALID_FILE_HANDLE
0x8983	IO_ERROR_NETWORK_DISK
0x8993	NO_READ_PRIVILEGES
0x8996	SERVER_OUT_OF_MEMORY
0x8998	VOLUME_DOES_NOT_EXIST
0x899B	BAD_DIRECTORY_HANDLE
0x899C	INVALID_PATH
0x899D	NO_MORE_DIRECTORY_HANDLES
0x89A1	DIRECTORY_IO_ERROR

Remarks

The directory handles allocated by NWAFPAllocTemporaryDirHandle are automatically deallocated when the task terminates.

NCP Calls

0x2222 35 11 AFP Alloc Temporary Directory Handle

See Also

NWAllocTemporaryDirectoryHandle, NWAllocTempNSDirHandle2 (Multiple and Inter-File Services)

NWAFPASCIIZToLenStr

Changes a NULL-terminated string to a length-preceded string

NetWare Server:

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: AFP

Syntax

```
#include <nwafp.h>
or
#include <nwcalls.h>

NWCCODE NWAPI NWAFPASCIIZToLenStr (
   pnstr8              pbstrDstStr,
   const nstr8 N_FAR  *pbstrSrcStr);
```

Delphi Syntax

```
uses calwin32

Function NWAFPASCIIZToLenStr
  (pbstrDstStr : pnstr8;
   const pbstrSrcStr : pnstr8
) : NWCCODE;
```

Parameters

pbstrDstStr

   (OUT) Points to a length-preceded string of Delphi type.

pbstrSrcStr

   (IN) Points to a NULL-terminated string.

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL

Remarks

NWAFPASCIIZToLenStr returns the length of the string if it is greater than the predetermined accepted size.

NWAFPCreateDirectory

Creates an AFP directory

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: AFP

Syntax

```
#include <nwafp.h>
or
#include <nwcalls.h>

NWCCODE NWAPI  NWAFPCreateDirectory  (
   NWCONN_HANDLE    conn,
   nuint16          volNum,
   nuint32          AFPEntryID,
   pnuint8          finderInfo,
   pnstr8           AFPPathString,
   pnuint32         newAFPEntryID);
```

Delphi Syntax

```
uses calwin32

Function NWAFPCreateDirectory
 (conn : NWCONN_HANDLE;
  volNum : nuint16;
  AFPEntryID : nuint32;
  finderInfo : pnuint8;
  AFPPathString : pnstr8;
  newAFPEntryID : pnuint32
) : NWCCODE;
```

Parameters

conn

  (IN) Specifies the NetWare server connection handle.

volNum

  (IN) Specifies the volume number of the directory entry location.

AFPEntryID

  (IN) Specifies the AFP base ID.

finderInfo

> (IN) Points to AFPFILEINFO containing the finder information for the new directory.

AFPPathString

> (IN) Points to the AFP directory path relative to `AFPEntryID.`

newAFPEntryID

> (OUT) Points to the ID of the newly created directory.

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x8805	NET_RECV_ERROR
0x890A	NLM_INVALID_CONNECTION
0x8983	IO_ERROR_NETWORK_DISK
0x8984	NO_CREATE_PRIVILEGES
0x8988	INVALID_FILE_HANDLE
0x8993	NO_READ_PRIVILEGES
0x8996	SERVER_OUT_OF_MEMORY
0x8998	VOLUME_DOES_NOT_EXIST
0x8999	DIRECTORY_FULL
0x899C	INVALID_PATH
0x899E	INVALID_FILENAME
0x89A1	DIRECTORY_IO_ERROR
0x89A2	READ_FILE_WITH_RECORD_LOCKED
0x89FD	BAD_STATION_NUMBER

NCP Calls

0x2222 35 13 AFP 2.0 Create Directory

See Also

NWAFPDelete (page 32), NWCreateDirectory (Multiple and Inter-File Services)

NWAFPCreateFile

Creates an AFP file

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: AFP

Syntax

```
#include <nwafp.h>
or
#include <nwcalls.h>

NWCCODE NWAPI  NWAFPCreateFile  (
   NWCONN_HANDLE    conn,
   nuint16          volNum,
   nuint32          AFPEntryID,
   nuint8           delExistingFile,
   pnuint8          finderInfo,
   pnstr8           AFPPathString,
   pnuint32         newAFPEntryID);
```

Delphi Syntax

```
uses calwin32

Function NWAFPCreateFile
  (conn : NWCONN_HANDLE;
   volNum : nuint16;
   AFPEntryID : nuint32;
   delExistingFile : nuint8;
   finderInfo : pnuint8;
   AFPPathString : pnstr8;
   newAFPEntryID : pnuint32
) : NWCCODE;
```

Parameters

conn

　　(IN) Specifies the NetWare server connection handle.

volNum

　　(IN) Specifies the volume number of the directory's entry location.

AFPEntryID

(IN) Specifies the AFP base ID.

delExistingFile

(IN) Specifies whether to delete the file of the same name (0 = do not delete).

finderInfo

(IN) Points to AFPFILEINFO containing the finder information for the new file.

AFPPathString

(IN) Points to the AFP directory path relative to `AFPEntryID`.

newAFPEntryID

(OUT) Points to the ID of the newly created directory.

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x8805	NET_RECV_ERROR
0x890A	NLM_INVALID_CONNECTION
0x8980	ERR_LOCK_FAIL
0x8981	NO_MORE_FILE_HANDLES
0x8983	IO_ERROR_NETWORK_DISK
0x8984	NO_CREATE_PRIVILEGES
0x8987	WILD_CARDS_IN_CREATE_FILE_NAME
0x8988	INVALID_FILE_HANDLE
0x898A	NO_DELETE_PRIVILEGES
0x898D	SOME_FILES_AFFECTED_IN_USE
0x898E	NO_FILES_AFFECTED_IN_USE
0x898F	SOME_FILES_AFFECTED_READ_ONLY
0x8990	NO_FILES_AFFECTED_READ_ONLY
0x8993	NO_READ_PRIVILEGES
0x8996	SERVER_OUT_OF_MEMORY
0x8998	VOLUME_DOES_NOT_EXIST
0x8999	DIRECTORY_FULL
0x899B	BAD_DIRECTORY_HANDLE
0x899C	INVALID_PATH

0x899E	INVALID_FILENAME
0x89A1	DIRECTORY_IO_ERROR
0x89A2	READ_FILE_WITH_RECORD_LOCKED
0x89FD	BAD_STATION_NUMBER
0x89FF	NO_FILES_FOUND_ERROR
0x89FF	File Exists Error

Remarks

The file resulting from NWAFPCreateFile is not opened; it is created as a normal Read/Write file with the system and hidden bits cleared.

For `AFPPathString`, byte 0 must be the length of the file name. The file name begins at byte 1 of the string. (Only include the file name—not the full path name—when calling NWAFPCreateFile.)

NCP Calls

0x2222 35 14 AFP 2.0 Create File

See Also

NWOpenNSEntry, NWOpenDataStream (Multiple and Inter-File Services), NWAFPDelete (page 32), NWAFPRename (page 49)

NWAFPDelete

Deletes an AFP file or directory

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: AFP

Syntax

```
#include <nwafp.h>
or
#include <nwcalls.h>

NWCCODE NWAPI NWAFPDelete (
   NWCONN_HANDLE       conn
   nuint16             volNum,
   nuint32             AFPEntryID,
   const nstr8 N_FAR  *AFPPathString);
```

Delphi Syntax

```
uses calwin32

Function NWAFPDelete
  (conn : NWCONN_HANDLE;
   volNum : nuint16;
   AFPEntryID : nuint32;
   const AFPPathString : pnstr8
) : NWCCODE;
```

Parameters

conn

   (IN) Specifies the NetWare server connection handle.

volNum

   (IN) Specifies the volume number of the directory entry location.

AFPEntryID

   (IN) Specifies the AFP base ID.

AFPPathString

   (IN) Points to the AFP directory path relative to AFPEntryID.

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x8805	NET_RECV_ERROR
0x890A	NLM_INVALID_CONNECTION
0x8983	IO_ERROR_NETWORK_DISK
0x8988	INVALID_FILE_HANDLE
0x898A	NO_DELETE_PRIVILEGES
0x898D	SOME_FILES_AFFECTED_IN_USE
0x898E	NO_FILES_AFFECTED_IN_USE
0x898F	SOME_FILES_AFFECTED_READ_ONLY
0x8990	NO_FILES_AFFECTED_READ_ONLY
0x8993	NO_READ_PRIVILEGES
0x8996	SERVER_OUT_OF_MEMORY
0x8998	VOLUME_DOES_NOT_EXIST
0x899B	Bad AFP Entry ID
0x899C	INVALID_PATH
0x899E	INVALID_FILENAME
0x899F	DIRECTORY_ACTIVE
0x89A0	DIRECTORY_NOT_EMPTY
0x89A1	DIRECTORY_IO_ERROR
0x89A2	READ_FILE_WITH_RECORD_LOCKED
0x89FD	BAD_STATION_NUMBER
0x89FF	NO_FILES_FOUND_ERROR
0x89FF	File Exists Error

Remarks

The directories to be deleted must be empty. Files to be deleted must be closed by all users.

For `AFPPathString`, byte 0 must be the length of the file name. The file name begins at byte 1 of the string. Include only the file name—not the full path name—when calling NWAFPDelete.

NCP Calls

0x2222 35 03 AFP Delete

See Also

NWAFPCreateDirectory (page 27), NWAFPCreateFile (page 29), NWAFPGetEntryIDFromName (page 39), NWAFPGetEntryIDFromHandle (page 37), NWAFPGetEntryIDFromPathName (page 41)

NWAFPDirectoryEntry

Tests a directory entry to see if it is an AFP file

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: AFP

Syntax

```
#include <nwafp.h>
or
#include <nwcalls.h>

NWCCODE NWAPI NWAFPDirectoryEntry  (
   NWCONN_HANDLE         conn,
   NWDIR_HANDLE          dirHandle,
   constr nstr8 N_FAR  *path);
```

Delphi Syntax

```
uses calwin32

Function NWAFPDirectoryEntry
  (conn : NWCONN_HANDLE;
   dirHandle : NWDIR_HANDLE;
   const path : pnstr8
) : NWCCODE;
```

Parameters

conn

    (IN) Specifies the NetWare server connection handle.

dirHandle

    (IN) Specifies the directory handle for the path name.

path

    (IN) Points to the path relative to `dirHandle`.

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	DOS File
0x0001	Macintosh File
0x8801	INVALID_CONNECTION
0x890A	NLM_INVALID_CONNECTION
0x8983	IO_ERROR_NETWORK_DISK
0x8988	INVALID_FILE_HANDLE
0x8993	NO_READ_PRIVILEGES
0x8996	SERVER_OUT_OF_MEMORY
0x899C	INVALID_PATH
0x89A2	READ_FILE_WITH_RECORD_LOCKED

Remarks

The `dirHandle` and `path` parameters must be given in the DOS name space format.

NCP Calls

0x2222 22 5 Get Volume Number
0x2222 22 21 Get Volume Info With Handle
0x2222 23 17 Get File Server Information
0x2222 23 22 Get Station's Logged Info (old)
0x2222 23 28 Get Station's Logged Info
0x2222 35 12 AFP Get Entry ID From Path Name
0x2222 35 15 AFP 2.0 Get File
0x2222 87 06 Obtain File or Subdirectory Information
0x2222 104 1 Ping for NDS NCP

See Also

NWAFPGetEntryIDFromPathName (page 41), NWGetVolumeInfoWithHandle (Volume Management), NWParsePath (Multiple and Inter-File Services), NWGetVolumeNumber (Volume Management), NWAFPGetFileInformation (page 43), NWGetOwningNameSpace (Multiple and Inter-File Services)

NWAFPGetEntryIDFromHandle

Returns an AFP entry ID for the specified NetWare handle

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: AFP

Syntax

```
#include <nwcaldef.h>
or
#include <nwcalls.h>

NWCCODE NWAPI NWAFPGetEntryIDFromHandle (
   NWCONN_HANDLE         conn,
   const nuint8 N_FAR   *NWHandle,
   pnuint16              volNum,
   pnuint32              AFPEntryID,
   pnuint8               forkIndicator);
```

Delphi Syntax

```
uses calwin32

Function NWAFPGetEntryIDFromHandle
  (conn : NWCONN_HANDLE;
   const NWHandle : pnuint8;
   volNum : pnuint16;
   AFPEntryID : pnuint32;
   forkIndicator : pnuint8
) : NWCCODE;
```

Parameters

conn

    (IN) Specifies the NetWare server connection handle.

NWHandle

    (IN) Points to the 6-byte NetWare handle for the path name.

volNum

    (OUT) Points to the volume number of the directory entry location.

AFPEntryID

(OUT) Points to the AFP file entry ID.

forkIndicator

(OUT) Points to the fork indicator (0 = data; 1 = resource).

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x890A	NLM_INVALID_CONNECTION
0x8988	INVALID_FILE_HANDLE
0x8998	VOLUME_DOES_NOT_EXIST
0x899C	INVALID_PATH

Remarks

`AFPEntryID` points to the AFP file ID. It is not the AFP base ID. INVALID_PATH will be returned if you use the `AFPEntryID` as the AFP base ID.

NCP Calls

0x2222 35 06 AFP Get Entry ID From NetWare Handle

See Also

NWAFPGetEntryIDFromName (page 39), NWAFPGetEntryIDFromPathName (page 41), NWAFPGetFileInformation (page 43), NWAFPAllocTemporaryDirHandle (page 24)

NWAFPGetEntryIDFromName

Returns a unique AFP entry ID from an AFP entry ID of a parent and a modifying path

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: AFP

Syntax

```
#include <nwafp.h>
or
#include <nwcalls.h>

NWCCODE NWAPI NWAFPGetEntryIDFromName (
   NWCONN_HANDLE        conn,
   nuint16              volNum,
   nuint32              AFPEntryID,
   const nstr8 N_FAR    *AFPPathString,
   pnuint32             newAFPEntryID);
```

Delphi Syntax

```
uses calwin32

Function NWAFPGetEntryIDFromName
  (conn : NWCONN_HANDLE;
   volNum : nuint16;
   AFPEntryID : nuint32;
   const AFPPathString : pnstr8;
   newAFPEntryID : pnuint32
) : NWCCODE;
```

Parameters

conn

   (IN) Specifies the NetWare server connection handle.

volNum

   (IN) Specifies the volume number of the directory entry location.

AFPEntryID

   (IN) Specifies the unique AFP base ID.

AFPPathString

(IN) Points to the path string modifying `AFPEntryID`.

newAFPEntryID

(OUT) Points to the AFP entry ID of the given path.

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x890A	NLM_INVALID_CONNECTION
0x8988	INVALID_FILE_HANDLE
0x8983	IO_ERROR_NETWORK_DISK
0x8993	NO_READ_RRIVILEGES
0x8996	SERVER_OUT_OF_MEMORY
0x899C	INVALID_PATH
0x8998	VOLUME_DOES_NOT_EXIST
0x899C	INVALID_PATH
0x89A1	DIRECTORY_IO_ERROR
0x89A2	READ_FILE_WITH_RECORD_LOCKED
0x89FD	BAD_STATION_NUMBER
0x89FF	NO_FILES_FOUND_ERROR

NCP Calls

0x2222 35 04 AFP Get Entry ID From Name

See Also

NWAFPGetEntryIDFromHandle (page 37), NWAFPGetEntryIDFromPathName (page 41)

NWAFPGetEntryIDFromPathName

Returns a unique 32-bit AFP file or directory ID, given a combination of path and directory handle

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: AFP

Syntax

```
#include <nwafp.h>
or
#include <nwcalls.h>

NWCCODE NWAPI  NWAFPGetEntryIDFromPathName  (
   NWCONN_HANDLE        conn,
   NWDIR_HANDLE         dirHandle,
   const nstr8 N_FAR   *path,
   pnuint32             AFPEntryID);
```

Delphi Syntax

```
uses calwin32

Function NWAFPGetEntryIDFromPathName
  (conn : NWCONN_HANDLE;
   dirHandle : NWDIR_HANDLE;
   const path : pnstr8;
   AFPEntryID : pnuint32
) : NWCCODE;
```

Parameters

conn

   (IN) Specifies the NetWare server connection handle.

dirHandle

   (IN) Specifies the directory handle for `path`.

path

   (IN) Points to the path given relative to the directory handle.

AFPEntryID

   (OUT) Points to the AFP base ID.

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x890A	NLM_INVALID_CONNECTION
0x8983	IO_ERROR_NETWORK_DISK
0x8988	INVALID_FILE_HANDLE
0x8993	NO_READ_RRIVILEGES
0x8996	SERVER_OUT_OF_MEMORY
0x8998	VOLUME_DOES_NOT_EXIST
0x899B	BAD_DIRECTORY_HANDLE
0x899C	INVALID_PATH
0x89A1	DIRECTORY_IO_ERROR
0x89A2	READ_FILE_WITH_RECORD_LOCKED
0x89FD	BAD_STATION_NUMBER
0x89FF	NO_FILES_FOUND_ERROR

Remarks

The directory base and path specifications must be given in DOS name space format.

NCP Calls

0x2222 35 12 AFP Get Entry ID From Path Name

See Also

NWAFPGetEntryIDFromHandle (page 37), NWAFPGetEntryIDFromName (page 39), NWAFPGetFileInformation (page 43), NWAFPAllocTemporaryDirHandle (page 24), NWAllocTempNSDirHandle2 (Multiple and Inter-File Services)

NWAFPGetFileInformation

Returns AFP information for a directory or file

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: AFP

Syntax

```
#include <nwafp.h>
or
#include <nwcalls.h>

NWCCODE NWAPI NWAFPGetFileInformation (
   NWCONN_HANDLE            conn,
   nuint16                  volNum,
   nuint32                  AFPEntryID,
   nuint16                  reqMask,
   const nstr8 N_FAR        *AFPPathString,
   nuint16                  structSize,
   NW_AFP_FILE_INFO N_FAR   *AFPFileInfo);
```

Delphi Syntax

```
uses calwin32

Function NWAFPGetFileInformation
  (conn : NWCONN_HANDLE;
   volNum : nuint16;
   AFPEntryID : nuint32;
   reqMask : nuint16;
   const AFPPathString : pnstr8;
   structSize : nuint16;
   Var AFPFileInfo : NW_AFP_FILE_INFO
) : NWCCODE;
```

Parameters

conn

   (IN) Specifies the NetWare server connection handle.

volNum

   (IN) Specifies the volume number of the directory entry location.

AFPEntryID

(IN) Specifies the unique AFP base ID.

reqMask

(IN) Specifies the request bit mask information.

AFPPathString

(IN) Points to the AFP directory path relative to `AFPEntryID`.

structSize

(IN) Specifies the request AFPFILEINFO buffer size.

AFPFileInfo

(OUT) Points to AFPFILEINFO returning AFP file information.

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x890A	NLM_INVALID_CONNECTION
0x8988	INVALID_FILE_HANDLE
0x8983	IO_ERROR_NETWORK_DISK
0x8993	NO_READ_PRIVILEGES
0x8996	SERVER_OUT_OF_MEMORY
0x8998	VOLUME_DOES_NOT_EXIST
0x899C	INVALID_PATH
0x89A1	DIRECTORY_I/O_ERROR
0x89A2	READ_FILE_WITH_RECORD_LOCKED
0x89FD	BAD_STATION_NUMBER
0x89FF	Failure. NO_FILES_FOUND_ERROR

Remarks

Valid bit map information request values follow for `reqMask` : (Bits can be ORed together.)

C Values	Delphi Values	Value Names
0x0001	$0001	AFP_GET_ATTRIBUTES
0x0002	$0002	AFP_GET_PARENT_ID
0x0004	$0004	AFP_GET_CREATE_DATE

C Values	Delphi Values	Value Names
0x0008	$0008	AFP_GET_ACCESS_DATE
0x0010	$0010	AFP_GET_MODIFY_DATETIME
0x0020	$0020	AFP_GET_BACKUP_DATETIME
0x0040	$0040	AFP_GET_FINDER_INFO
0x0080	$0080	AFP_GET_LONG_NAME
0x0100	$0100	AFP_GET_ENTRY_ID
0x0200	$0200	AFP_GET_DATA_LEN
0x0400	$0400	AFP_GET_RESOURCE_LEN
0x0800	$0800	AFP_GET_NUM_OFFSPRING
0x1000	$1000	AFP_GET_OWNER_ID
0x2000	$2000	AFP_GET_SHORT_NAME
0x4000	$4000	AFP_GET_ACCESS_RIGHTS
0x8000	$8000	AFP_GET_PRO_DOS_INFO
0xffff	$ffff	AFP_GET_ALL

NCP Calls

0x2222 35 15 AFP 2.0 Get File

See Also

NWAFPSetFileInformation (page 56), NWAFPScanFileInformation (page 52)

NWAFPOpenFileFork

Opens an AFP file fork from a DOS environment

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: AFP

Syntax

```
#include <nwafp.h>
or
#include <nwcalls.h>

NWCCODE NWAPI NWAFPOpenFileFork (
   NWCONN_HANDLE        conn,
   nuint16              volNum,
   nuint32              AFPEntryID,
   nuint8               forkIndicator,
   nuint8               accessMode,
   const nstr8 N_FAR   *AFPPathString,
   pnuint32             fileID,
   pnuint32             forkLength,
   pnuint8              NWHandle,
  NWFILE_HANDLE N_FAR  *DOSFileHandle);
```

Delphi Syntax

```
uses calwin32

Function NWAFPOpenFileFork
  (conn : NWCONN_HANDLE;
   volNum : nuint16;
   AFPEntryID : nuint32;
   forkIndicator : nuint8;
   accessMode : nuint8;
   const AFPPathString : pnstr8;
   fileID : pnuint32;
   forkLength : pnuint32;
   NWHandle : pnuint8;
   Var DOSFileHandle : NWFILE_HANDLE
) : NWCCODE;
```

Parameters

conn

(IN) Specifies the NetWare server connection handle.

volNum

(IN) Specifies the volume number of the directory entry location.

AFPEntryID

(IN) Specifies the AFP base ID.

forkIndicator

(IN) Specifies the data or resource fork indicator:

0 Data
1 Resource

accessMode

(IN) Specifies the file access mode indicator.

AFPPathString

(IN) Points to the AFP directory path relative to AFPEntryID.

fileID

(OUT) Points to the file entry ID.

forkLength

(OUT) Points to the length of the opened fork.

NWHandle

(OUT) Points to the 6-byte NetWare file handle.

DOSFileHandle

(OUT) Points to the file handle.

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x890A	NLM_INVALID_CONNECTION
0x8980	FILE_IN_USE_ERROR
0x8981	NO_MORE_FILE_HANDLES
0x8988	INVALID_FILE_HANDLE
0x8983	IO_ERROR_NETWORK_DISK

0x8993	NO_READ_PRIVILEGES
0x8994	NO_WRITE_PRIVILEGES_OR_READONLY
0x8996	SERVER_OUT_OF_MEMORY
0x8998	VOLUME_DOES_NOT_EXIST
0x8999	DIRECTORY_FULL
0x899C	Invalid AFP Path String
0x89A1	DIRECTORY_IO_ERROR
0x89A2	READ_FILE_WITH_RECORD_LOCKED
0x89FD	BAD_STATION_NUMBER
0x89FF	LOCK_ERROR, NO_FILES_FOUND_ERROR

Remarks

If a file does not exist, NWAFPOpenFileFork returns NO_FILES_FOUND_ERROR.

If an existing file does not have a resource or data file fork associated with it, NWAFPOpenFileFork will automatically create and open the specified file fork.

accessMode can have the following values:

C Value	Delphi Value	Name
0x0001	$0001	AR_READ and AR_READ_ONLY
0x0002	$0002	AR_WRITE and AR_WRITE_ONLY
0x0004	$0004	AR_DENY_READ
0x0008	$0008	AR_DENY_WRITE
0x0010	$0010	AR_COMPATIBILITY

NCP Calls

0x2222 35 08 AFP Open File Fork

See Also

NWAFPCreateFile (page 29), NWAFPGetFileInformation (page 43), NWAFPGetEntryIDFromName (page 39)

NWAFPRename

Renames an AFP file

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: AFP

Syntax

```
#include <nwafp.h>
or
#include <nwcalls.h>

NWCCODE NWAPI NWAFPRename (
   NWCONN_HANDLE        conn,
   nuint16              volNum,
   nuint32              AFPSourceEntryID,
   nuint32              AFPDestEntryID,
   const nstr8 N_FAR  *AFPSrcPath,
   const nstr8 N_FAR  *AFPDstPath);
```

Delphi Syntax

```
uses calwin32

Function NWAFPRename
  (conn : NWCONN_HANDLE;
   volNum : nuint16;
   AFPSourceEntryID : nuint32;
   AFPDestEntryID : nuint32;
   const AFPSrcPath : pnstr8;
   const AFPDstPath : pnstr8
) : NWCCODE;
```

Parameters

conn

   (IN) Specifies the NetWare server connection handle.

volNum

   (IN) Specifies the volume number of the directory entry location.

AFPSourceEntryID

   (IN) Specifies the AFP source base ID.

AFPDestEntryID

   (IN) Specifies the AFP destination base ID.

AFPSrcPath

   (IN) Points to the AFP source directory path relative to `AFPSourceEntryID`.

AFPDstPath

   (IN) Points to the AFP destination directory path, relative to `AFPDestEntryID`.

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x890A	NLM_INVALID_CONNECTION
0x8984	NO_CREATE_PRIVILEGES
0x8988	INVALID_FILE_HANDLE
0x8983	IO_ERROR_NETWORK_DISK
0x898B	NO_RENAME_PRIVILEGES
0x898E	NO_FILES_AFFECTED_IN_USE
0x8990	NO_FILES_AFFECTED_READ_ONLY
0x8992	NO_FILES_RENAMED_NAME_EXISTS
0x8993	NO_READ_PRIVILEGES
0x8996	SERVER_OUT_OF_MEMORY
0x8998	VOLUME_DOES_NOT_EXIST
0x8999	DIRECTORY_FULL
0x899C	Invalid AFP Path String
0x899E	INVALID_FILENAME
0x89A1	DIRECTORY_IO_ERROR
0x89A2	READ_FILE_WITH_RECORD_LOCKED
0x89FD	BAD_STATION_NUMBER
0x89FF	NO_FILES_FOUND_ERROR

NCP Calls

0x2222 35 07 AFP Rename

See Also

NWNSRename (Multiple and Inter-File Services), NWAFPGetEntryIDFromName (page 39)

NWAFPScanFileInformation

Scans a directory and returns AFP file/directory information

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: AFP

Syntax

```
#include <nwafp.h>
or
#include <nwcalls.h>

NWCCODE NWAPI NWAFPScanFileInformation (
   NWCONN_HANDLE            conn,
   nuint16                  volNum,
   nuint32                  AFPEntryID,
   pnuint32                 AFPLastSeenID,
   nuint16                  searchMask,
   nuint16                  reqMask,
   const nstr8 N_FAR      *AFPPathString,
   nuint16                  structSize,
   NW_AFP_FILE_INFO N_FAR  *AFPFileInfo);
```

Delphi Syntax

```
uses calwin32

Function NWAFPScanFileInformation
  (conn : NWCONN_HANDLE;
   volNum : nuint16;
   AFPEntryID : nuint32;
   AFPLastSeenID : pnuint32;
   searchMask : nuint16;
   reqMask : nuint16;
   const AFPPathString : pnstr8;
   structSize : nuint16;
   Var AFPFileInfo : NW_AFP_FILE_INFO
) : NWCCODE;
```

Parameters

conn

> (IN) Specifies the NetWare server connection handle.

volNum

(IN) Specifies the volume number of the directory entry location.

AFPEntryID

(IN) Specifies the AFP base ID.

AFPLastSeenID

(IN) Points to `AFPEntryID`.

searchMask

(IN) Specifies the search mask.

reqMask

(IN) Specifies the request bit mask information.

AFPPathString

(IN) Points to the AFP directory path relative to `AFPEntryID`.

structSize

(IN) Specifies the size of the AFPFILEINFO buffer.

AFPFileInfo

(OUT) Points to AFPFILEINFO returning AFP file information.

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x890A	NLM_INVALID_CONNECTION
0x8983	IO_ERROR_NETWORK_DISK
0x8988	INVALID_PATH
0x8993	NO_READ_PRIVILEGES
0x8996	SERVER_OUT_OF_MEMORY
0x8998	VOLUME_DOES_NOT_EXIST
0x899C	INVALID_PATH
0x89A1	DIRECTORY_IO_ERROR
0x89A2	READ_FILE_WITH_RECORD_LOCKED
0x89FD	BAD_STATION_NUMBER
0x89FF	NO_FILES_FOUND_ERROR

Remarks

AFPLastSeenID should be initialized to -1 on the first iteration.

Valid bit map information request values follow for reqMask. (Bits can be ORed together.)

C Values	Delphi Values	Value Names
0x0001	$0001	AFP_GET_ATTRIBUTES
0x0002	$0002	AFP_GET_PARENT_ID
0x0004	$0004	AFP_GET_CREATE_DATE
0x0008	$0008	AFP_GET_ACCESS_DATE
0x0010	$0010	AFP_GET_MODIFY_DATETIME
0x0020	$0020	AFP_GET_BACKUP_DATETIME
0x0040	$0040	AFP_GET_FINDER_INFO
0x0080	$0080	AFP_GET_LONG_NAME
0x0100	$0100	AFP_GET_ENTRY_ID
0x0200	$0200	AFP_GET_DATA_LEN
0x0400	$0400	AFP_GET_RESOURCE_LEN
0x0800	$0800	AFP_GET_NUM_OFFSPRING
0x1000	$1000	AFP_GET_OWNER_ID
0x2000	$2000	AFP_GET_SHORT_NAME
0x4000	$4000	AFP_GET_ACCESS_RIGHTS
0x8000	$8000	AFP_GET_PRO_DOS_INFO
0xffff	$ffff	AFP_GET_ALL

NWSEARCH_MASK is defined as follows:

C Values	Delphi Values	Value Names
0x0000	$0000	AFP_SA_NORMAL
0x0100	$0100	AFP_SA_HIDDEN
0x0200	$0200	AFP_SA_SYSTEM
0x0400	$0400	AFP_SA_SUBDIR
0x0800	$0800	AFP_SA_FILES
0xF00	$0F00	AFP_SA_ALL

AFPFILEINFO attributes follow:

```
0x0001 = Search Mode
0x0002 = Search Mode
```

```
0x0004 = Search Mode
0x0008 = Undefined
0x0010 = Transaction
0x0020 = Index
0x0040 = Read Audit
0x0080 = Write Audit
0x0100 = Read Only
0x0200 = Hidden
0x0400 = System
0x0800 = Execute Only
0x1000 = Subdirectory
0x2000 = Archive
0x4000 = Undefined
0x8000 = Shareable File
```

NCP Calls

0x2222 35 17 AFP 2.0 Scan File Information

See Also

NWAFPGetFileInformation (page 43)

NWAFPSetFileInformation

Sets AFP information for a file or directory

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: AFP

Syntax

```
#include <nwafp.h>
or
#include <nwcalls.h>

NWCCODE NWAPI NWAFPSetFileInformation (
   NWCONN_HANDLE          conn,
   nuint16                volNum,
   nuint32                AFPBaseID,
   nuint16                reqMask,
   const nstr8 N_FAR      *AFPPathString,
   nuint16                structSize,
   NW_AFP_SET_INFO N_FAR  *AFPSetInfo);
```

Delphi Syntax

```
uses calwin32

Function NWAFPSetFileInformation
  (conn : NWCONN_HANDLE;
   volNum : nuint16;
   AFPBaseID : nuint32;
   reqMask : nuint16;
   const AFPPathString : pnstr8;
   structSize : nuint16;
   Var AFPSetInfo : NW_AFP_SET_INFO
) : NWCCODE;
```

Parameters

conn

   (IN) Specifies the NetWare server connection handle.

volNum

   (IN) Specifies the volume number of the directory entry location.

AFPBaseID

(IN) Specifies the AFP base ID.

reqMask

(IN) Specifies the request bit mask information.

AFPPathString

(IN) Points to the AFP directory path relative to `AFPBaseID`.

structSize

(IN) Specifies the size of the AFPSETINFO buffer.

AFPSetInfo

(IN) Points to AFPSETINFO to set AFP file information.

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x8901	ERR_INSUFFICIENT_SPACE
0x890A	NLM_INVALID_CONNECTION
0x8983	IO_ERROR_NETWORK_DISK
0x8988	INVALID_FILE_HANDLE
0x8993	NO_READ_PRIVILEGES
0x8994	NO_WRITE_PRIVILEGES_OR_READONLY
0x8995	FILE_DETACHED
0x8996	SERVER_OUT_OF_MEMORY
0x8998	VOLUME_DOES_NOT_EXIST
0x899C	INVALID_PATH
0x89A1	DIRECTORY_IO_ERROR
0x89A2	READ_FILE_WITH_RECORD_LOCKED
0x89FD	BAD_STATION_NUMBER
0x89FF	Failure; NO_FILES_FOUND_ERROR

Remarks

The following constants are used by NWAFPSetFileInformation to manipulate `requestMask`. They are also used in by `NWSEARCH_MASK` in NWAFPScanFileInformation.

C Values	Delphi Values	Value Names
0x0001	$0001	AFP_GET_ATTRIBUTES
0x0002	$0002	AFP_GET_PARENT_ID
0x0004	$0004	AFP_GET_CREATE_DATE
0x0008	$0008	AFP_GET_ACCESS_DATE
0x0010	$0010	AFP_GET_MODIFY_DATETIME
0x0020	$0020	AFP_GET_BACKUP_DATETIME
0x0040	$0040	AFP_GET_FINDER_INFO
0x0080	$0080	AFP_GET_LONG_NAME
0x0100	$0100	AFP_GET_ENTRY_ID
0x0200	$0200	AFP_GET_DATA_LEN
0x0400	$0400	AFP_GET_RESOURCE_LEN
0x0800	$0800	AFP_GET_NUM_OFFSPRING
0x1000	$1000	AFP_GET_OWNER_ID
0x2000	$2000	AFP_GET_SHORT_NAME
0x4000	$4000	AFP_GET_ACCESS_RIGHTS
0x8000	$8000	AFP_GET_PRO_DOS_INFO
0xffff	$ffff	AFP_GET_ALL

These constants identify AFP entries to be included in NWAFPSetFileInformation.

C Values	Delphi Values	Value Names
0x0000	$0000	AFP_SA_NORMAL
0x0100	$0100	AFP_SA_HIDDEN
0x0200	$0200	AFP_SA_SYSTEM
0x0400	$0400	AFP_SA_SUBDIR
0x0800	$0800	AFP_SA_FILES
0xF00	$0F00	AFP_SA_ALL

Valid bit map information request values follow for reqMask: (Bits can be ORed together.)

C Values	Delphi Values	Value Names
0x0001	$0001	AFP_SET_ATTRIBUTES
0x0004	$0004	AFP_SET_CREATE_DATE
0x0008	$0008	AFP_SET_ACCESS_DATE

C Values	Delphi Values	Value Names
0x0010	$0010	AFP_SET_MODIFY_DATETIME
0x0020	$0020	AFP_SET_BACKUP_DATETIME
0x0040	$0040	AFP_SET_FINDER_INFO
0x8000	$8000	AFP_SET_PRO_DOS_INFO

AFPSETINFO `attributes` follow:

```
0x0001 = Search Mode
0x0002 = Search Mode
0x0004 = Search Mode
0x0008 = Undefined
0x0010 = Transaction
0x0020 = Index
0x0040 = Read Audit
0x0080 = Write Audit
0x0100 = Read Only
0x0200 = Hidden
0x0400 = System
0x0800 = Execute Only
0x1000 = Subdirectory
0x2000 = Archive
0x4000 = Undefined
0x8000 = Shareable File
```

NCP Calls

0x2222 35 16 AFP 2.0 Set File Information

See Also

(Multiple and Inter-File Services)

NWAFPSupported

Reports whether the AFP is supported on a server volume

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: AFP

Syntax

```
#include <nwafp.h>
or
#include <nwcalls.h>

NWCCODE NWAPI  NWAFPSupported  (
   NWCONN_HANDLE   conn,
   nuint16         volNum);
```

Delphi Syntax

```
uses calwin32

Function NWAFPSupported
  (conn : NWCONN_HANDLE;
   volNum : nuint16
) : NWCCODE;
```

Parameters

conn

    (IN) Specifies the NetWare server connection handle.

volNum

    (IN) Specifies the volume number to test.

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	AFP supported
Nonzero	AFP not supported

NCP Calls

0x2222 22 6 Get Volume Name
0x2222 35 12 AFP Get Entry ID From Path Name

See Also

NWReadNSInfo (Multiple and Inter-File Services), NWGetVolumeName (Volume Management), NWAFPGetEntryIDFromPathName (page 41)

AFP Structures

4

This documentation alphabetically lists the AFP structures and describes their purpose, syntax, and fields.

NW_AFP_FILE_INFO

Defines file information for AFP files

Service: AFP

Defined In: nwafp.h

Structure

```
typedef struct
{
   nuint32   entryID ;
   nuint32   parentID ;
   nuint16   attributes ;
   nuint32   dataForkLength ;
   nuint32   resourceForkLength ;
   nuint16   numOffspring ;
   nuint16   creationDate ;
   nuint16   accessDate ;
   nuint16   modifyDate ;
   nuint16   modifyTime ;
   nuint16   backupDate ;
   nuint16   backupTime ;
   nuint8    finderInfo [32];
   nstr8     longName [34];
   nuint32   ownerID ;
   nstr8     shortName [14];
   nuint16   accessPrivileges ;
   nuint8    proDOSInfo [6];
} NW_AFP_FILE_INFO, AFPFILEINFO;
```

Delphi Structure

```
uses calwin32

AFPFILEINFO = packed Record
   entryID : nuint32;
   parentID : nuint32;
   attributes : nuint16;
   dataForkLength : nuint32;
   resourceForkLength : nuint32;
   numOffspring : nuint16;
   creationDate : nuint16;
   accessDate : nuint16;
   modifyDate : nuint16;
   modifyTime : nuint16;
   backupDate : nuint16;
   backupTime : nuint16;
   finderInfo : Array[0..31] Of nuint8;
   longName : Array[0..33] Of nstr8;
   ownerID : nuint32;
   shortName : Array[0..13] Of nstr8;
```

```
   accessPrivileges : nuint16;
   proDOSInfo : Array[0..5] Of nuint8
End;
```

Fields

entryID

Specifies the unique AFP identifier of a file or directory.

parentID

Specifies the unique AFP identifier for the parent directory. The root will be 0.

attributes

Specifies the set of bits identifying the entry's attributes:

0x0001 = Search Mode
0x0002 = Search Mode
0x0004 = Search Mode
0x0008 = Undefined
0x0010 = Transaction
0x0020 = Index
0x0040 = Read Audit
0x0080 = Write Audit
0x0100 = Read Only
0x0200 = Hidden
0x0400 = System
0x0800 = Execute Only
0x1000 = Subdirectory
0x2000 = Archive
0x4000 = Undefined
0x8000 = Shareable File

dataForkLength

Specifies the data size of the target AFP file. If `pathModString` specifies an AFP directory, `dataForkLength` returns a zero (0).

resourceForkLength

Specifies the resource fork size of the target AFP file. If `pathModString` specifies an AFP directory, `resourceForkLength` returns a zero.

numOffspring

Specifies the number of files and subdirectories contained within the specified directory. If the AFP directory or file path specifies an AFP file, `numOffspring` returns a zero (0).

creationDate

Specifies the creation date (in AFP format) of the target directory or file.

accessDate

Specifies when the target AFP file was last accessed (returned in AFP format). If `pathModString` specifies an AFP directory, `accessDate` returns a zero.

modifyDate

Specifies the last modified date (in AFP format) of the target AFP file. If `pathModString` specifies an AFP directory, `modifyDate` returns zero.

modifyTime

Specifies the last modified time (in AFP format) of the target AFP file. If `pathModString` specifies an AFP directory, `modifyTime` returns zero.

backupDate

Specifies the last backup date (in AFP format) of the specified directory or file.

backupTime

Specifies the last backup time (in AFP format) of the specified directory or file.

finderInfo

Specifies the 32-byte finder information structure associated with each AFP directory or file.

longName

Specifies the AFP directory or file name of the specified directory or file. An AFP directory or file name can be from 1 to 31 characters long. `longName` is a null-terminated ASCII string. One extra byte has been added for the NULL terminator and another byte has been added to ensure word alignment.

ownerID

Specifies the 4-byte bindery object ID of the object creating or last modifying the file.

shortName

Specifies the NetWare® directory or file name of the specified directory or file in the DOS name space. A NetWare directory or file name is in DOS 8.3 format. `shortName` is a null-terminated ASCII string. One extra byte has been added for the NULL terminator and another byte has been added to ensure word alignment.

accessPrivileges

Specifies the one-word bit mask of the calling station's privileges for accessing the specified file or directory.

proDOSInfo

Specifies the 6-byte structure defined in Apple documentation.

NW_AFP_SET_INFO

Defines Apple file attributes

Service: AFP

Defined In: nwafp.h

Structure

```
typedef struct
{
   nuint16   attributes ;
   nuint16   creationDate ;
   nuint16   accessDate ;
   nuint16   modifyDate ;
   nuint16   modifyTime ;
   nuint16   backupDate ;
   nuint16   backupTime ;
   nuint8    finderInfo [32];
   nuint8    proDOSInfo [6];
} NW_AFP_SET_INFO, AFPSETINFO;
```

Delphi Structure

```
uses calwin32

AFPSETINFO = packed Record
   attributes : nuint16;
   creationDate : nuint16;
   accessDate : nuint16;
   modifyDate : nuint16;
   modifyTime : nuint16;
   backupDate : nuint16;
   backupTime : nuint16;
   finderInfo : Array[0..31] Of nuint8;
   proDOSInfo : Array[0..5] Of nuint8
 End;
```

Fields

attributes

> Specifies the file attributes.

creationDate

> Specifies the creation date (in AFP format) of the target directory or file.

accessDate

> Specifies when the target AFP file was last accessed (returned in AFP format).

modifyDate

Specifies the last modified date (in AFP format) of the target AFP file.

modifyTime

Specifies the last modified time (in AFP format) of the target AFP file.

backupDate

Specifies the last date (in AFP format) the file was backed up.

backupTime

Specifies the time (in AFP format) the file was last backed up.

finderInfo

Specifies the information defined in Apple documentation.

proDOSInfo

Specifies the 6-byte structure defined in Apple documentation.

RECPKT_AFPFILEINFO

Is the structure actually returned from the NCP call

Service: AFP

Defined In: nwafp.h

Structure

```
typedef struct
{
   nuint32   entryID ;
   nuint32   parentID ;
   nuint16   attributes ;
   nuint32   dataForkLength ;
   nuint32   resourceForkLength ;
   nuint16   numOffspring ;
   nuint16   creationDate ;
   nuint16   accessDate ;
   nuint16   modifyDate ;
   nuint16   modifyTime ;
   nuint16   backupDate ;
   nuint16   backupTime ;
   nuint8    finderInfo [32];
   nstr8     longName [32];
   nuint32   ownerID ;
   nstr8     shortName [12];
   nuint16   accessPrivileges ;
   nuint8    proDOSInfo [6];
} RECPKT_AFPFILEINFO;
```

Delphi Structure

```
uses calwin32

RECPKT_AFPFILEINFO = packed Record
   entryID : nuint32;
   parentID : nuint32;
   attributes : nuint16;
   dataForkLength : nuint32;
   resourceForkLength : nuint32;
   numOffspring : nuint16;
   creationDate : nuint16;
   accessDate : nuint16;
   modifyDate : nuint16;
   modifyTime : nuint16;
   backupDate : nuint16;
   backupTime : nuint16;
   finderInfo : Array[0..31] Of nuint8;
   longName : Array[0..31] Of nstr8;
   ownerID : nuint32;
   shortName : Array[0..11] Of nstr8;
```

```
accessPrivileges : nuint16;
proDOSInfo : Array[0..5] Of nuint8
End;
```

Fields

entryID

Specifies the unique AFP identifier of a file or directory.

parentID

Specifies the unique AFP identifier for the parent directory. The root will be 0.

attributes

Specifies the set of bits identifying the entry's attributes:

0x0001 = Search Mode
0x0002 = Search Mode
0x0004 = Search Mode
0x0008 = Undefined
0x0010 = Transaction
0x0020 = Index
0x0040 = Read Audit
0x0080 = Write Audit
0x0100 = Read Only
0x0200 = Hidden
0x0400 = System
0x0800 = Execute Only
0x1000 = Subdirectory
0x2000 = Archive
0x4000 = Undefined
0x8000 = Shareable File

dataForkLength

Specifies the data size of the target AFP file. If `pathModString` specifies an AFP directory, `dataForkLength` returns a zero (0).

resourceForkLength

Specifies the resource fork size of the target AFP file. If `pathModString` specifies an AFP directory, `resourceForkLength` returns a zero.

numOffspring

Specifies the number of files and subdirectories contained within the specified directory. If the AFP directory or file path specifies an AFP file, `numOffspring` returns a zero (0).

creationDate

Specifies the creation date (in AFP format) of the target directory or file.

accessDate

Specifies when the target AFP file was last accessed (returned in AFP format). If `pathModString` specifies an AFP directory, `accessDate` returns a zero.

`modifyDate`

Specifies the last modified date (in AFP format) of the target AFP file. If `pathModString` specifies an AFP directory, `modifyDate` returns zero.

`modifyTime`

Specifies the last modified time (in AFP format) of the target AFP file. If `pathModString` specifies an AFP directory, `modifyTime` returns zero.

`backupDate`

Specifies the last backup date (in AFP format) of the specified directory or file.

`backupTime`

Specifies the last backup time (in AFP format) of the specified directory or file.

`finderInfo`

Specifies the 32-byte finder information structure associated with each AFP directory or file.

`longName`

Specifies the AFP directory or file name of the specified directory or file. An AFP directory or file name can be from 1 to 31 characters long.

`ownerID`

Specifies the 4-byte bindery object ID of the object creating or last modifying the file.

`shortName`

Specifies the NetWare® directory or file name of the specified directory or file in the DOS name space. A NetWare directory or file name is in DOS 8.3 format.

`accessPrivileges`

Specifies the one-word bit mask of the calling station's privileges for accessing the specified file or directory.

`proDOSInfo`

Specifies the 6-byte structure defined in Apple documentation.

Direct File System Concepts

5

This documentation describes Direct File System, its functions, and features.

NetWare® Direct File System (DFS) functions provide a method of bypassing the NetWare disk caching and TTS subsystems. The NetWare cache and NetWare TTS provide the best possible throughput and should not normally be bypassed. However, several applications exist where this bypass might be desirable:

- Large database packages typically provide their own caching and transaction tracking facilities, tailored to specific requirements and integrated into the package, making it desirable to use their caching and transaction facilities instead of those provided by NetWare. (Performance degradation results when both are used together.)
- Backup applications often access large amounts of data not being accessed by other applications. If these accesses are made through the cache, the cache becomes non-relevant for all other accesses, and general server performance suffers for other users.
- Some utilities might require the ability to specify exactly where files are placed so that volumes can be defragmented and files accessed optimally.

The DFS functions provide a standardized solution by bypassing the NetWare disk caching and TTS subsystems, while providing more direct control of file allocation. The DFS functions fall into two groups: basic direct mode file I/O, and file allocation primitives.

The following figure illustrates the DFS as it interfaces with the file system, applications, NLM applications, and other portions of the NetWare OS:

Figure 5-1 *DFS Interface Flow Chart*
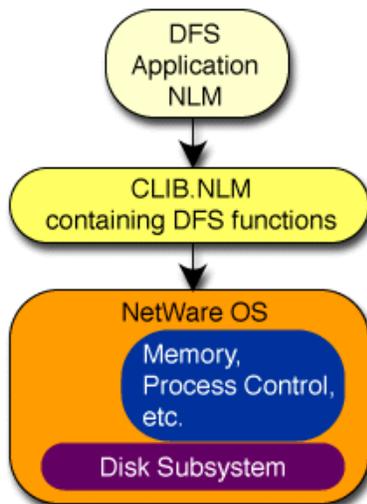
5.1 File Allocation

The DFS allocation functions allow a great deal of control in determining where and how file space is allocated. The application NLM might allow the OS to allocate required space for a file using OS default allocation, or might request allocation to be on a specific volume segment and/or specify the

actual volume blocks to be allocated. File allocation can be used to fill holes in sparse files or to extend existing files. Writing to a nonexistent area in a file (either to a hole or beyond the allocated file space) is not allowed with the DFS functions.

- "Impact of Striping" on page 74
- "Setting the File Size and Zero-Filling with DFS" on page 74

See the following related tasks as well:

5.1.1 Impact of Striping

Causing a file to be allocated with striping has different effects on different configurations. A drive array normally provides optimum throughput using internal striping, so specifying striping by the OS in this case would defeat optimum throughput, while specifying striping on files on a SCSI host adapter providing disconnect with multiple drives normally provides more optimal performance. Striping does not necessarily provide significant performance benefits for extremely large files accessed in true random fashion, but provides performance benefits when accessed sequentially.

5.1.2 Setting the File Size and Zero-Filling with DFS

The size of a file can be specified independent of the actual file space allocated by calling DFSSetEndOfFile (page 105). Reads attempted beyond the current end-of-file (indicated by the file size) are always rejected as an attempted operation beyond the current file size.

If the file size is expanded beyond its previous value, the additional file area incorporated in the new file size is zero-filled, provided that the file space is actually allocated. If the file size specified is smaller than the previously indicated file size and the `returnTrunctatedBlocksFlag` is nonzero, blocks previously allocated beyond the newly defined file size are truncated, that is, returned to the OS for future use.

The file size is also modified by calling DFSWrite (page 110) (only the "wait" version) indicating a write to a file sector address beyond the current file size (the file size is updated accordingly) but within the range of blocks allocated for the file. If the write does not start immediately following the current file size, the intervening blocks are zero-filled.

When a file has additional space allocated at the end of the file by calling DFSExpandFile (page 89), the additional file space is not zero-filled immediately. Subsequent writes to the file set a new file size and eliminate the requirement to zero-fill the additional file space. Calling DFSSetEndOfFile (page 105) also sets a new file size, and zero-fills any sectors in the new file size that are beyond the previous file size.

When a file has additional space allocated to fill a hole in a sparse file, the DFSExpandFile (page 89) function also zero-fills the additional space if the space is not beyond the current file size.

5.2 File Locks

All DFS application file, record, and field locks must be provided and managed by the DFS application NLM. OS utilities that are designed to perform reorganization or relocation of DFS files

use an exclusive lock on the file, so that the lock fails if the file is currently in use by an application NLM. An application NLM need only lock the file with a shared or nonexclusive lock to ensure that an OS utility NLM has not exclusively locked the file for operations such as reorganization.

An application NLM should make sure that the current connection ID matches the connection ID of the client so that OS Auditing is meaningful. However, auditing of database record and field accesses require that the database application NLM perform its own auditing, since DFS is not aware of the actual record and field definitions.

5.2.1 Input and Output

An open file is in one of two possible modes: normal mode or direct mode. In a normal open mode, any of the valid opens are used. For direct open mode, the DFSsopen (page 107) function is used. If a file is already open in normal mode and DFSsopen (page 107) is called, the file changes from the normal mode to the direct mode.

In this condition, any files in the normal open mode can continue to be read, but attempts to write to the file fail. Programs using the direct mode can read and write to the file successfully. The only way to write to files in the normal open mode again is by closing all direct mode opens, and closing and re-opening the normal mode open.

When a file is successfully opened for direct file I/O by calling DFSsopen (page 107), the server opens the file, flushes all cache entries for the file, and flags the file so that future I/Os do not use caching and TTS functions. Subsequent file I/O must be done using the I/O functions DFSRead (page 93), DFSReadNoWait (page 95), DFSWrite (page 110), or DFSWriteNoWait (page 112). The "no wait" versions of these functions do not block execution of the thread until completion, thus allowing a single thread to have multiple outstanding DFS I/O functions.

When direct file I/O operations are completed, the file must be closed by calling DFSclose (page 86). The file must not be used for normal I/O writes until all handles are relinquished for the file by calling DFSclose (page 86), followed by a normal open. If a normal open exists, it must be closed and reopened for read/writes.

When a file is in Direct mode, writes must not be made to areas of the file where a hole exists (sparse files) or beyond the file's currently allocated last block address. Read operations to these areas are indicated successful and the data area is zeroed.

Also, a file cannot be extended while in Direct mode by writing beyond its current extents. A separate call to DFSExpandFile (page 89) must be made to extend a file or to fill in holes in a file area.

5.3 File Structures

Files in direct file mode can be viewed as an array of sectors numbered from zero to *n*, where *n* is the last sector in the last block currently allocated for the file. All direct file I/O must be done in multiples of sectors (one or more). The sectors allocated in the file are actually allocated on an allocation block size, which is either 4 KB, 8 KB, 16 KB, 32 KB, or 64 KB with NetWare® (the default allocation block size for NetWare 3.x is 4 KB). The allocation size must be specified for a volume when the volume is created.

It is possible to create files that have blocks allocated for high addresses, but do not have blocks allocated for intermediate addresses. (Such files are called sparse files, and have holes in their actual allocation space). Normal (non-direct) I/O allows writes to be made into these holes or beyond the

current end of allocated space for a file, in which case space is automatically allocated to fill the hole or extend the file. However, the Direct File System does not allow such writes when the file is in direct mode. Files must be extended before writes can be issued to holes or beyond the end of the allocated space for a file.

The logical block address of the blocks in the above array are referred to as File Block Addresses, and the associated logical sector addresses are referred to as File Sector Addresses. These addresses exist even for holes in the file, though actual storage space might not have been allocated for the corresponding locations on the file.

No space is allocated when a file is initially created by DFScreat (page 87) or DFSsopen (page 107). All space must be allocated by the application process for files in direct mode (and must be allocated on the same volume).

When a file not currently opened by another process is opened in direct mode, the OS creates a turbo FAT for the file if one does not currently exist, flushes the cache of entries for the file, and marks the file as open in direct mode (only direct mode operations are allowed).

The following figures shows a file with 4 KB allocation block size and only one block allocated:

Figure 5-2 *One Allocated Block (4 KB)*
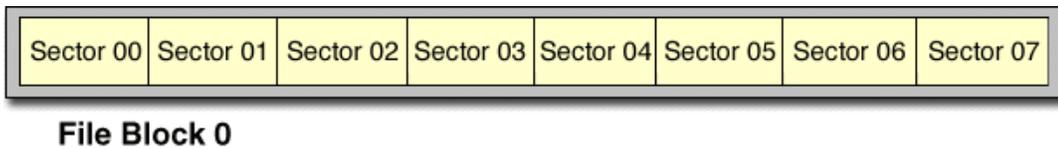
The following figure shows a file with 4 KB allocation block size and four blocks allocated (no holes):

Figure 5-3 *Four Allocated Blocks (4 KB each with no holes)*
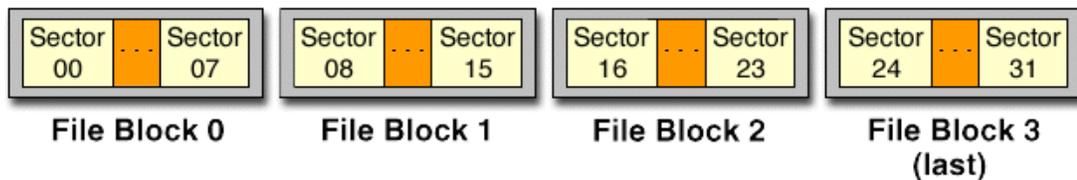
The following figure shows a file with 4 KB allocation block size and three blocks allocated (with hole):

Figure 5-4 *Three Allocated Blocks (4 KB each with hole)*
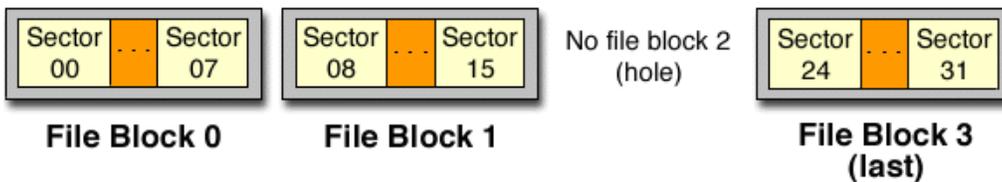
The following figure shows a file with 64 KB allocation and three blocks allocated (with hole):

Figure 5-5 *Three Allocated Blocks (64 KB each with hole)*
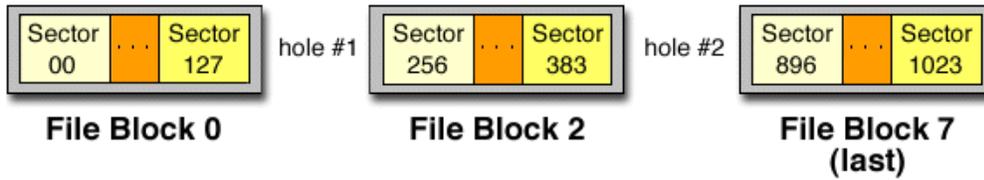
The following figure shows a file with 8 KB allocation block size and three blocks allocated (no holes):

Figure 5-6 *Three Allocated Blocks (8 KB each with no holes)*
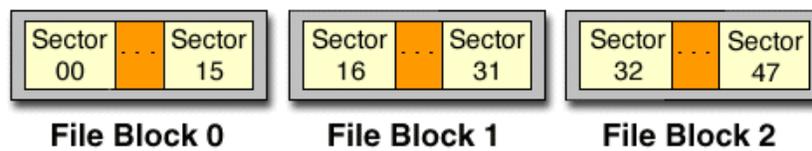
5.4 Volume Structures

NetWare® volumes can be viewed as an array of allocation blocks numbered from zero to *n*, where *n* is the total number of allocation blocks in all segments of the volume (volume segments), minus one. Volumes can be extended by adding additional segments, which are logically added at the end of the list of current volume segments. Each logical block number in the logical array of volume blocks is referred to by a *Volume Block Number*. There are no holes in the volume block numbers. (Using this organization, a specific volume block number also indirectly indicates the segment upon which it occurs in the list of volume segments.)

NetWare supports a maximum of 255 volumes, with a NetWare volume consisting of from 1 to 64 segments.

Multiple volume segments can exist on a single logical NetWare partition on a drive. It is also possible that a drive has a single volume segment on it. A Logical partition has blocks numbered logically from zero through the last block available in the partition, and does not include the physical partition blocks which are allocated for Hot Fix™ and Mirroring tables at the time the NetWare partition is created.

The following figure shows a volume with a single volume segment (9444 volume blocks):

Figure 5-7 *Single Volume Segment*
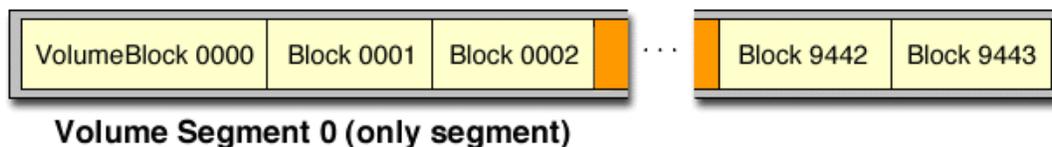
The following figure shows a volume with four volume segments (25288 volume blocks):

Figure 5-8 *Four Volume Segments*

Volume Segment 0 (9444 blocks)

Volume Segment 1 (3200 blocks)

Volume Segment 2 (9444 blocks)

Volume Segment 3 (3200 blocks)
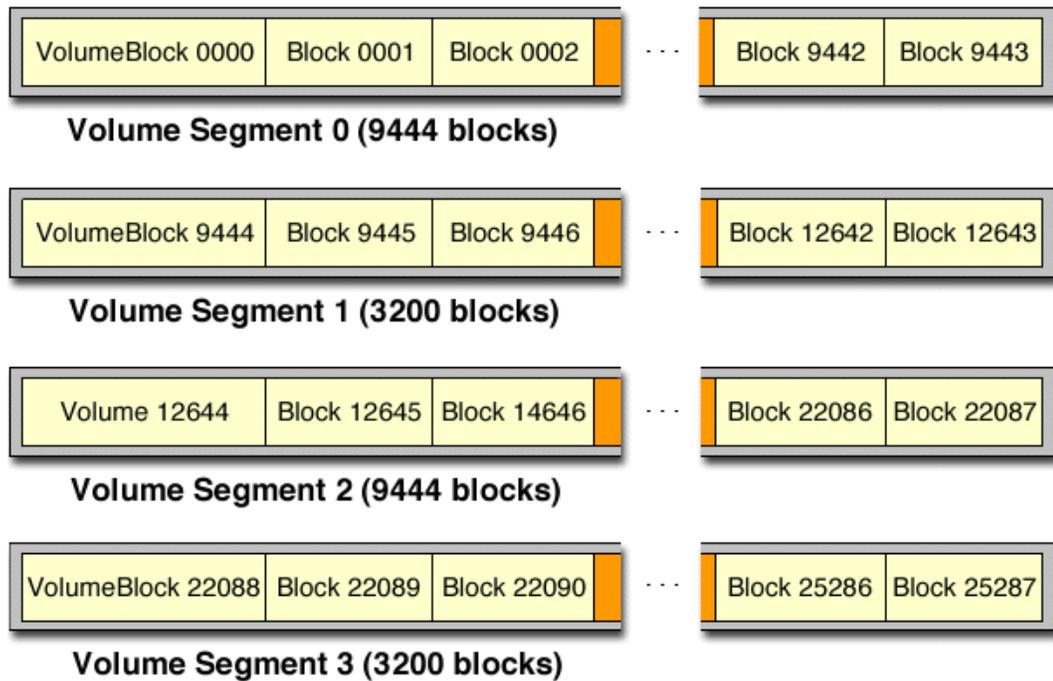
5.5 Return Values

The following table lists and defines DFS return values:

Decimal	Constant	Description
0	DFSNormalCompletion	The operation was completed as specified.
1	DFSInsufficientSpace	The required space does not currently exist on the volume to expand the file as requested.
4	DFSVolumeSegmentDeactivated	The volume segments on which the file is located have been deactivated by the operating system.
16	DFSTruncationFailure	DFSSetEndOfFile (page 105) detected that the caller requested excess blocks to be truncated, but the file was open for other connections, causing the request to be rejected.
17	DFSHoleInFileError	An operation (read or write) was requested for one or more file sectors where file space has not been allocated (a sparse file). The buffer is also zeroed for read functions.
18	DFSParameterError	The function caller supplied an invalid parameter.
19	DFSOverlapError	An attempt was made to allocate additional file space where file blocks already exist.

Decimal	Constant	Description
20	DFSSegmentError	A volume segment number was requested that was not one of the volume segments of the volume.
21	DFSBoundaryError	One or more blocks in the range requested are not currently available, or are not in the volume or volume segment specified.
22	DFSInsufficientLimboFileSpace	The request could not be completed because there were not enough contiguous limbo blocks to complete the request successfully.
23	DFSNotInDirectFileMode	A function requiring a file to be opened in direct mode (using DFSsopen (page 107) or DFScreat (page 87)) was requested, but the file is not currently opened in direct mode.
24	DFSOperationBeyondEndOfFile	A read operation was requested beyond the end-of-file (current file size).
129	DFSOutOfHandles	All available handles for the file are already in use.
131	DFSHardIOError	Problem decompressing or insufficient allocatable space.
136	DFSInvalidFileHandle	A DFS function was called using a file handle that is not valid—typically an open was omitted or the user inadvertently closed the file before attempting this access.
147	DFSNoReadPrivilege	The current connection does not have read privileges for the file.
148	DFSNoWritePrivilege	The current connection does not have write privileges for the file.
149	DFSFileDetached	The file system will not allow further processing on this handle.
150	DFSInsufficientMemory	The Direct File System could not obtain sufficient memory to complete the requested function.
152	DFSInvalidVolume	The volume number specified does not exist or not mounted.
-1	DFSFailedCompletion	The requested operation was not completed.

5.6 Direct File System Functions

These functions allow you to directly manage file systems:

DFSclose	Closes a file that is open in direct file mode.
DFScreat	Creates and opens a file in direct file mode, and returns a file handle.
DFSExpandFile	Expands a file with a range of contiguous blocks.

DFSFreeLimboVolumeSpace	Frees a number of limbo blocks on a volume.
DFSRead	Reads sectors from a file in direct file mode (sleeps until completion).
DFSReadNoWait	Reads sectors from a file in direct file mode (returns immediately after initiation).
DFSReturnFileMappingInformation	Returns file extents, each with number of blocks and starting file and volume block numbers.
DFSReturnVolumeBlockInformation	Returns volume block usage bitmap.
DFSReturnVolumeMappingInformation	Returns information about a volume required for file allocation.
DFSSetEndOfFile	Sets the file size of a file.
DFSsopen	Opens a file in direct file mode.
DFSWrite	Writes sectors into a file using DFS (sleeps until completion).
DFSWriteNoWait	Writes sectors into a file using DFS (returns immediately after initiation).

Direct File System Tasks

<div style="text-align: right; font-size: 3em;">6</div>

This documentation describes common tasks associated with Direct File System.

6.1 Creating a File

Files can be created by calling DFScreat (page 87). This creates a file on the specified volume with the permissions indicated, and leaves the file open in Direct File Mode (the file must later be closed). DFScreat (page 87) is the DFS equivalent of creat (page 207).

Create files by calling DFSsopen (page 107) (providing that the file does not currently exist). DFSsopen (page 107) is the DFS equivalent of sopen (page 238).

6.2 Extending Files Using Allocation

Extend an NLM application file using default file allocation:

1 Determine if space is available.

Call DFSReturnVolumeMappingInformation (page 101) to obtain the volume size, allocation unit size, sector size, number of free blocks, number of blocks available in deleted files, number of blocks not available in deleted files, and so forth. If the desired number of blocks is not available on the volume, skip to Step 4.

Since the OS is multitasking, it is possible (and likely) that other processes allocate (and free) volume blocks at any time, making information returned by DFSReturnVolumeMappingInformation (page 101) obsolete. As a result you must design DFS NLM applications to repeat this procedure multiple times to deal with the failure of DFSExpandFile (page 89) and DFSFreeLimboVolumeSpace (page 91).

2 Determine the current allocation of the file, specifically the file block where the file is to be extended.

Call DFSReturnFileMappingInformation (page 97) to determine the file's current mapping, including volume segments and blocks allocated. The starting file block to be extended must be specified in the extend request made in Step 3.

A file might not be extended by specifying file block addresses that already exist. This means that an extend request to allocate file space for a hole in a sparse file must not specify a number or contiguous blocks that exceed the size of the hole.

3 Allocate blocks to expand the file.

Call DFSExpandFile (page 89) specifying the number of blocks to be extended and wildcards (-1) for the volume block number and optionally for the volume segment number. This allows DFS to select the range of contiguous blocks used to extend the file.

For normal files, extending a file a single block at a time and specifying wildcards for *both* the volume block address and the volume segment is identical in function to letting the NetWare® OS allocate for normal files.

Specifying a wildcard segment number in conjunction with a wildcard volume block address allows the OS to alternate its selection of volume segments for file allocation, thus facilitating file striping on systems where it is advantageous to stripe files. Specifying a larger number of

blocks with a wildcard volume segment stripes the file (provided that multiple volume segments exist) with the larger granularity. If there is not enough contiguous available space to expand the file by the requested number of blocks, applications may be required to make several calls specifying smaller request sizes, or fail the request.

If the return code indicates that the above operation was not successful, proceed to Step 4. Otherwise, the file has been extended as requested.

4 Free up volume space.

Call DFSFreeLimboVolumeSpace (page 91) specifying the volume number and the requested number of blocks to be freed. This causes one or more deleted files to be purged from the volume in order of time of deletion. Go to Step 1.

6.3 Extending Files Using Specific Allocation

Extend an NLM application file by selecting the volume segments and/or blocks to be allocated:

1 Determine if space is available.

Call DFSReturnVolumeMappingInformation (page 101) to obtain the volume size, allocation unit size, sector size, number of free blocks, number of blocks available in deleted files, number of blocks not available in deleted files, and so forth. If adequate space for the file extension is not available, go to Step 5.

2 Determine the current allocation for the file.

Call DFSReturnFileMappingInformation (page 97) to determine the file's current mapping, including volume segments and blocks allocated. This information is required to determine where to extend a file.

A file may not be extended by specifying file block addresses which already exist. This means that an extend request to allocate file space for a hole in a sparse file must not specify a number or contiguous blocks which exceed the size of the hole.

3 Determine the available blocks on a volume.

Call DFSReturnVolumeBlockInformation (page 99) to obtain a bit map of available blocks on the desired volume. Determine from the bitmap a contiguous range of blocks large enough to extend the file as needed.

As a result of multitasking, the bitmap of available blocks is valid only at the moment it is obtained, and may have changed by the time an application NLM requests a specific range of blocks to be allocated for a file (possibly requiring this step to be repeated multiple times).

Call DFSExpandFile (page 89) to specify the range of contiguous available blocks selected from the preceding bitmap. Specifying more than a single block causes the requested contiguous blocks to be allocated from the same volume segment. A good return code indicates that the requested function is completed.

4 Extend the file.

If enough contiguous space to expand the file the requested number of blocks is not available, applications may be required to make several calls specifying smaller request sizes, or fail the request.

If the return code indicates that the above operation was not successful, proceed to Step 5. Some other module may have allocated the requested blocks, so the calling application must be prepared to retry the operation several times.

5 Free up volume space.

Call DFSFreeLimboVolumeSpace (page 91) to specify the volume number and the requested number of blocks to be freed. This causes one or more deleted files to be purged from the volume in the order of the time of deletion. Go back to Step 1.

Direct File System Functions

7

This documentation alphabetically lists the direct file system functions and describes their purpose, syntax, parameters, and return values.

DFSclose

Closes a file currently open in Direct File Mode

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x, 5.x, 6.x

Platform: NLM

Service: Direct File System

Syntax

```
#include <nwdfs.h>

LONG DFSclose  (
   LONG   fileHandle);
```

Parameters

fileHandle

> (IN) Specifies the file handle returned from a prior successful call to DFSsOpen (the file must have previously opened by DFSsOpen). After a file is closed, the file handle is no longer valid and should not be reused.

Return Values

Decimal	Hex	Status Code	Description
0	(0x00)	DFSNormalCompletion	The operation was completed as specified.
-1		DFSFailedCompletion	An error occurred closing the file. If this status is returned, `errno` is set to: 4 EBADF (Bad file number). If the function does not complete successfully, `NetWareErrno` is set.

Remarks

Calling DFSclose causes the file to be closed (the handle becomes invalid). If DFSClose determines that this was the last valid handle (no other opens outstanding for the file), the Direct File Mode flag is reset, allowing a subsequent open file call to be either open (normal mode), or DFSsOpen (direct mode). Remember, if a file is in direct mode, any programs with normal mode opens into the file are able to read the file but not write to it (see).

See Also

DFScreat

Creates and opens a file in Direct File Mode, returning a file handle to the called file

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x, 5.x, 6.x

Platform: NLM

Service: Direct File System

Syntax

```
#include <nwdfs.h>

LONG DFScreat  (
   BYTE   *fileName,
   LONG    access,
   LONG    flagBits);
```

Parameters

fileName

(IN) Points to the name of the file to be created. The filename must be NULL-terminated and must include the path, including the volume name but not the server name.

access

(IN) Specifies the access permissions for the file.

flagBits

(IN) Specifies the following when a file is created:

0x0001   DELETE_FILE_ON_CREATE_BIT
0x0002   NO_RIGHTS_CHECK_ON_CREATE_BIT

Return Values

!= -1	The file now exists, is open, and is in direct mode. The return value is the file handle assigned when the file was created.
== -1	An error occurred creating the file.

If -1 is returned, `errno` is set to

Decimal	Constant	Description
1	ENONENT	No such file.

Decimal	Constant	Description
6	EACCES	Permission denied.
9	EINVAL	Invalid argument.

If the function does not complete successfully, `NetWareErrno` is set to

Decimal	Hex	Constant
152	(0x98)	ERR_INVALID_VOLUME
156	(0x9C)	ERR_INVALID_PATH

Remarks

Calling DFScreat causes DFS to create a file, or to truncate the file if it already exists *and* if the current connection has write privileges. The name of the file to be created is given by the `filename` parameter. If the file exists, it is truncated to contain no data and the preceding permission setting is unchanged. The file is switched to direct mode, forcing subsequent file accesses to be direct (The file must be extended using DFSExpandFile to provide required file space). The file is left open and must be closed by a subsequent call to DFSclose.

Not all functions are allowed with this form of open once the file has been created. If additional functions such as specifying a stream are required, the caller should close the file and open it again by calling DFSsopen.

The `access` permissions are defined in FCNTL.H as follows:

Hex	Constant	Description
0x0000	O_RDONLY	open for read only now if this wrap
0x0001	O_WRONLY	open for write only
0x0002	O_RDWR	open for read and write
0x0010	O_APPEND	writes done at end of file
0x0020	O_CREAT	create new file if one does not exist
0x0040	O_TRUNC	truncate existing file
0x0080	O_EXCL	exclusive open

If `access` is 0, the default value is O_CREAT, O_TRUNC, and O_WRONLY.

See Also

DFSclose (page 86), DFSExpandFile (page 89), DFSsopen (page 107)

DFSExpandFile

Requests DFS to expand a file with a range of contiguous blocks

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x, 5.x, 6.x

Platform: NLM

Service: Direct File System

Syntax

```
#include <nwdfs.h>

LONG DFSExpandFile  (
   LONG   fileHandle,
   LONG   fileBlockNumber,
   LONG   numberOfBlocks,
   LONG   volumeBlockNumber,
   LONG   segmentNumber);
```

Parameters

fileHandle

   (IN) Specifies the file handle returned from a prior DFSsopen or DFScreat call.

fileBlockNumber

   (IN) Specifies the beginning file logical block number where the additional contiguous space is to be allocated.

numberOfBlocks

   (IN) Specifies the number of contiguous blocks requested to be linked into the file allocation at the starting location.

volumeBlockNumber

   (IN) Specifies the beginning volume logical block number at which contiguous blocks are to be allocated for the file. A wildcard value of -1 indicates that DFS can allocate the blocks anywhere it can find the required contiguous space on the volume segment.

segmentNumber

   (IN) Specifies the volume segment number where the contiguous blocks are to be allocated when the logical volume block number is not specified (a wildcard volume block number was provided). A wildcard value of -1 in this parameter indicates that DFS can allocate the blocks in any volume segment on the volume where the specified number of contiguous free blocks can be found.

Return Values

Decimal	Constant	Description
0	DFSNormalCompletion	File was expanded in the area specified.
1	DFSInsufficientSpace	The required space does not exist on the volume to expand the file as requested.
18	DFSParameterError	The caller supplied one or more invalid parameters.
19	DFSOverlapError	An attempt was made to allocate additional file space where file blocks already exist.
20	DFSSegmentError	A volume segment was specified which does not exist on the volume.
21	DFSBoundaryError	One or more blocks in the range requested are not available or are not in the volume or volume segment specified.
131	DFSHardIOError	Attempted allocation of file blocks where file blocks are already defined, etc.
136	DFSInvalidFileHandle	A DFS call was made using a file handle which is not valid—typically an open was omitted or the user inadvertently closed the file before attempting this access.
148	DFSNoWritePrivilege	The current connection does not have write privileges for this file.
149	DFSFileDetached	The function was not performed because the file is detached.

Remarks

The DFSExpandFile function is required to expand a file or to write in a hole in a sparse file (DFSWrite and DFSWriteNoWait cannot expand a file by writing beyond the current end of the file or by writing in a hole in a sparse file. Also, normal (non-direct) writes which could normally expand a file are rejected by the OS while a file is in direct file mode).

Since it is always possible that DFS might find that some of the blocks in the indicated range have been allocated by other threads or processes after the caller determined that they were free, the caller must handle this contingency. It is logical that the caller repeat the sequence of freeing limbo blocks and attempting to expand several times before reducing the number of contiguous blocks requested and making multiple requests. For details on striping and other allocation details, see "Impact of Striping" on page 74. New file space allocated to fill a hole in a sparse file is zero-filled. Contiguous blocks added to the end of allocated file space are not zero-filled.

NOTE: A range of blocks that spans two volume segments is not considered contiguous, even though the logical volume block addresses are contiguous.

See Also

DFSFreeLimboVolumeSpace

Requests DFS to free a number of limbo blocks on a volume

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x, 5.x, 6.x

Platform: NLM

Service: Direct File System

Syntax

```
#include <nwdfs.h>

LONG DFSFreeLimboVolumeSpace  (
   LONG   volumeNumber,
   LONG   numberOfBlocks);
```

Parameters

volumeNumber

   (IN) Specifies the volume number where the requested number of limbo blocks are to be freed.

numberOfBlocks

   (IN) Specifies the number of limbo blocks requested to be freed.

Return Values

Decimal	Constant	Description
0	DFSNormalCompletion	The operation was completed as specified.
22	DFSInsufficientLimboFileSpace	The request could not be completed because there were not enough contiguous limbo blocks to complete the request successfully.
152	DFSInvalidVolume	The volume does not exist or is not mounted.

Remarks

This function requests the OS to free a number of limbo blocks on a given volume. This function performs the equivalent of a purge of one or more files until it has freed the requested number of blocks (or more). There is no guarantee that the OS can free as many blocks as requested by the caller, or that the blocks freed are contiguous. Also there is no way to guarantee that the blocks will be made available on a specific volume segment.

Other processes, including system functions, can acquire blocks that have just been freed before they can be allocated. The OS normally stripes allocation of files when multiple segments exist for a

volume, so it can be very difficult to find a large contiguous area of free blocks on a volume where non-direct or normal files are allocated in multisegment volumes. Callers should be prepared to call this function multiple times.

See Also

DFSExpandFile (page 89)

DFSRead

Reads the sectors requested from a file using DFS (sleeps until completion)

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x, 5.x, 6.x

Platform: NLM

Service: Direct File System

Syntax

```
#include <nwdfs.h>

LONG DFSRead  (
   LONG    fileHandle,
   LONG    startingSector,
   LONG    sectorCount,
   BYTE   *buffer);
```

Parameters

fileHandle

   (IN) Specifies the file handle returned from a prior call to open for the indicated file.

startingSector

   (IN) Specifies the starting sector number (logical offset from beginning of the file) in the file where the read operation is to begin.

sectorCount

   (IN) Specifies the number of sectors to be read into the buffer.

buffer

   (OUT) Points to a contiguous buffer area large enough to contain the number of sectors indicated to be read.

Return Values

Decimal	Constant	Description
0	DFSNormalCompletion	The operation was completed as specified.
17	DFSHoleInFileError	A read was attempted to file sector addresses where no file blocks are allocated. The buffer is zero-filled.
18	DFSParameterError	The caller supplied one or more invalid parameters.

Decimal	Constant	Description
23	DFSNotInDirectFileMode	A direct file read (DFSRead) was issued but the file has not been opened successfully in direct mode.
24	DFSOperationBeyondEndOfFile	A read function was requested beyond the current end of file.
131	DFSHardIOError	
136	DFSInvalidFileHandle	A DFS call was made using a file handle which is not valid. Typically an open was omitted or the user inadvertently closed the file before attempting this access.
147	DFSNoReadPrivilege	Current connection does not have read privileges for the file.
149	DFSFileDetached	The function was not completed because the file is detached.
150	DFSInsufficientMemory	DFS could not allocate sufficient memory to complete the request.
162	DFSIOLockError	

Remarks

This function performs a read of one or more sectors using a logical zero-based sector offset into the indicated file. Since this function is blocking, control is returned to the caller after all reads relating to the requested read function have been completed. If a status indicating a hole was detected during the requested read operation, the buffer is zeroed. It is not possible to read beyond the end of the allocated area of a file.

See Also

DFSReadNoWait

Reads the sectors requested from a file using DFS (returns after initiation)

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x, 5.x, 6.x

Platform: NLM

Service: Direct File System

Syntax

```
#include <nwdfs.h>

LONG DFSReadNoWait  (
   LONG      fileHandle,
   LONG      startingSector,
   LONG      sectorCount,
   BYTE     *buffer,
   struct DFSCallBackParameters   *callBackNode);
```

Parameters

fileHandle

   (IN) Specifies the file handle returned from a prior call to open for the indicated file.

startingSector

   (IN) Specifies the starting sector number (logical offset from beginning of file) in the file where the read operation is to begin.

sectorCount

   (IN) Specifies the number of sectors to be read into the buffer.

buffer

   (OUT) Points to a contiguous buffer area large enough to contain the number of sectors indicated to be read.

callBackNode

   (IN) Pointer to a structure used to signal completion of all requested reads for a particular call to DFSReadNoWait.

Return Values

0	Read Normal Initiation
!=0	Read not initiated

> **NOTE:** The actual completion is stored in the completionCode field of the DFSCallBackParameters upon completion of the request.

Remarks

This function is identical to DFSRead, except that the return to the function caller is made immediately after posting the reads to the driver. This means that the status returned from the function only indicates whether the call was initiated or not. The completion status is returned in the structure provided for completion notification. A calling process must allow other processes to run. Consequently, any long sequence of code including this function call should make frequent calls to ThreadSwitch to allow other processes to be executed.

The DFSCallBackParameters structure is defined as follows:

```
struct DFSCallBackParameters
{
   LONG   localSemaphoreHandle;
   LONG   completionCode;
};
```

The `localSemaphoreHandle` field contains a local semaphore handle obtained by calling OpenLocalSemaphore. WaitOnLocalSemaphore or ExamineLocalSemaphore should be called to determine when the semaphore has been signalled.

The `completionCode` field contains the actual completion code, initialized to -1 by this function and updated upon completion. For completion code values, see DFSRead.

See Also

DFSRead (page 93), DFSWriteNoWait (page 112), OpenLocalSemaphore (*NDK: NLM Threads Management*)

DFSReturnFileMappingInformation

Returns file extents, each with the number of blocks, and starting file and volume block numbers

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 4.x, 5.x, 6.x

Platform: NLM

Service: Direct File System

Syntax

```
#include <nwdfs.h>

LONG DFSReturnFileMappingInformation  (
   LONG    fileHandle,
   LONG    startingBlockNumber,
   LONG   *numberOfEntries,
   LONG    tableSize,
   struct FileMapStructure   *table);
```

Parameters

fileHandle

   (IN) Specifies the file handle returned from a prior call to open for the file.

startingBlockNumber

   (IN) Specifies the starting file block address for which map is requested (zero relative).

numberOfEntries

   (OUT) Points to the number of valid file map entries returned.

tableSize

   (IN) Specifies the number of file map entries for which space has been allocated by the caller in the following table (that is, max # of struct FileMapStructure to be returned).

table

   (OUT) Points to a table of file map entries.

Return Values

Decimal	Constant	Description
0	DFSNormalCompletion	The operation was completed and information fields are valid.
18	DFSParameterError	The function caller supplied an invalid parameter.

Decimal	Constant	Description
136	DFSInvalidFileHandle	A DFS call was made using a file handle that is not valid. Typically an open was omitted or the user inadvertently closed the file before attempting this access.

Remarks

This function is required to provide the calling NLM application with details of exactly where a given file's logical blocks are located, including where file holes and the end of a file's allocated storage space is, so that the application can expand the file by calling DFSExpandFile.

See Also

DFSReturnVolumeBlockInformation

Returns the volume block usage bitmap for requested volume

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 4.x, 5.x, 6.x

Platform: NLM

Service: Direct File System

Syntax

```
#include <nwdfs.h>

LONG DFSReturnVolumeBlockInformation  (
   LONG    volumeNumber,
   LONG    startingBlockNumber,
   LONG    numberOfBlocks,
   BYTE   *buffer);
```

Parameters

volumeNumber

   (IN) Specifies the volume number for which the volume block information is desired.

startingBlockNumber

   (IN) Specifies volume logical block zero or an even multiple of 8 up to the last block of the volume.

numberOfBlocks

   (IN/OUT) Specifies the number of blocks for which allocation bit flags are to be transferred into the buffer (If `startingBlockNumber` plus `numberOfBlocks` is greater than the total number of volume blocks, the number of volume blocks remaining starting from `startingBlockNumber` is substituted here).

buffer

   (OUT) Points to a pointer to a buffer area where the information is returned. The area required for the buffer is the number of blocks rounded up modulus 8. The format of the data in the buffer is bit array, with 1 bits indicating available blocks. The relative bit address of each bit is the block address relative to the beginning of the specified starting file block number.

Return Values

Decimal	Constant	Description
0	DFSNormalCompletion	The operation is complete and information fields are valid.

Decimal	Constant	Description
152	DFSInvalidVolume	The volume number specified does not exist or not mounted.

Remarks

This function is used to determine which blocks on a volume are in use and that are available for allocation. This function returns a bitmap which has a bit for each block in the range specified in the calling parameters, beginning with the logical (zero-based) volume block indicated by `startingBlockNumber`. This information is required if an application NLM is attempting to do specific allocation for a file, in order to pick block ranges of contiguous free blocks to expand a file.

The data returned by this function is only valid until it is changed by some request, and can change dynamically before an application can successfully request allocation of the blocks selected. The application process must be designed to handle this exception, as well as the case where there is not a single contiguous free block area large enough to satisfy the file expansion request.

See Also

DFSReturnFileMappingInformation (page 97), DFSReturnVolumeMappingInformation (page 101)

DFSReturnVolumeMappingInformation

Returns information about a volume required for file allocation

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 4.x, 5.x, 6.x

Platform: NLM

Service: Direct File System

Syntax

```
#include <nwdfs.h>

LONG DFSReturnVolumeMappingInformation  (
   LONG                                   volumeNumber,
   struct VolumeInformationStructure   *volumeInformation);
```

Parameters

volumeNumber

(IN) Specifies the system volume number of the selected volume. A requesting process can determine the appropriate volume number using the File System Services functions.

volumeInformation

(OUT) Points to a structure of type VolumeInformationStructure (see below).

Return Values

Decimal	Constant	Description
0	DFSNormalCompletion	The operation is complete and information fields are valid.
152	DFSInvalidVolume	The volume number specified does not exist or is not mounted.

Remarks

This function provides volume information, including allocation block size, that is necessary to determine how many blocks to allocate in order to allocate a specified number of bytes for a request. This information changes dynamically. Therefore, it can become invalid before an application NLM can use the information. Application NLM applications must be designed to handle the likelihood that blocks available change dynamically.

See Also

DFSSetDataSize

Sets file size

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 5.x, 6.x

Platform: NLM

Service: Direct File System

Syntax

```
#include <nwdfs.h>

LONG DFSSetDataSize  (
   LONG               handle,
   unsigned__int64    newFileSize,
   LONG               setSizeFlags);
```

Parameters

handle

   (IN) Specifies the handle returned from DFSsopen or DFScreat for the file on which to set the size

newFileSize

   (IN) Specifies the logical byte offset at which the new end of the file is to be set

setSizeFlags

   (IN) Specifes a bit in a mask indicating the ways in which the file can be expanded

Return Values

Zero on success or nonzero if an error occurs.

Remarks

DFSSetDataSize modifies the data size (end of file) for the open file identified by `handle`. If the file has more than one data stream, only the size of the data stream identified by `handle` is modified. To modify the size of more than one data stream, an application must open each stream independently and call DFSSetDataSize once for each data stream.

DFSSetDataSize works with the Novell Storage Services file system only. If `handle` specifies a file for any other system, DFSSetDataSize will not operate.

If the new size is smaller than the original size, the data at the end of the original data stream is truncated. If the new size is larger than the original size, the data size is expanded.

The `setSizeFlags` parameter provides some control over file expansion through the following bits:

SETSIZE_NON_SPARSE_FILE	If a file is being expanded with this bit set, it is expanded in a nonsparce manner, and data is physically allocated to the file. Blocks in such added data are zero-filled unless SETSIZE_NO_ZERO_FILL is also set. If this bit is not set the file is expanded in a sparce manner and no data is physically written to the file.
SETSIZE_NO_ZERO_FILL	With this bit set, if a file is being expanded in a nonsparce manner, newly allocated disk blocks are not zero filled. The blocks are allocated to the file but not initialized. If this bit is not set, zero-filled data is physically written from the original end of the file to the new end.
	This bit is allowed only from the NLM application running on the local server and cannot be set by an NCP operation.
SETSIZE_UNDO_ON_ERR	If an error occurs during file expansion, an error is returned and the file is restored to its original size as though no expansion had taken place. If this bit is not set and an error occurs during expansion, the remaining expansion is aborted, and any partially completed expansion remains part of the file.
SETSIZE_PHYSICAL_ONLY	Only the physical end of the file is changed. The logical end is untouched. File expansion with this bit set is done in a nonsparce manner as though SETSIZE_NON_SPARSE_FILE were set. Truncation is likewise only physical, with no change to the logical end of the file.
SETSIZE_LOGICAL_ONLY	With this bit set, DFSSetDataSize changes only the logical end of the file. Allocated physical storage is neither expanded nor truncated, but remains unchanged.

See Also

DFScreat (page 87), DFSSetEndOfFile (page 105), DFSsopen (page 107)

DFSSetEndOfFile

Sets file size

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x, 5.x, 6.x

Platform: NLM

Service: Direct File System

Syntax

```
#include <nwdfs.h>

LONG DFSSetEndOfFile  (
   LONG   fileHandle,
   LONG   newFileSize,
   LONG   returnTruncatedBlocksFlag);
```

Parameters

fileHandle

　　(IN) Specifies the file handle returned from a prior DFSsopen or DFScreat call for the indicated file.

newFileSize

　　(IN) Specifies the new file size in bytes.

returnTruncatedBlocksFlag

　　(IN) Specifies a nonzero value specified here indicates that any blocks truncated by the new file size should be freed up for future OS use.

Return Values

Decimal	Constant	Description
0	DFSNormalCompletion	The operation is complete as specified.
16	DFSTruncationFailure	The DFSSetEndOfFile function detected that the caller requested excess blocks to be truncated, but the file was open for other connections, causing the request to be rejected.
131	DFSHardIOError	
136	DFSInvalidFileHandle	A call was made with an invalid file handle. Typically an open was omitted or the user inadvertently closed the file.

Remarks

If the connection making the request is the only entity with the file open and if the `returnTruncatedBlocksFlag` is nonzero, setting a new file size that is one or more blocks less than the previous file size causes blocks (actually allocated) beyond the new defined file size to be truncated, or returned for future OS usage. If a new file size is specified that is greater than the previous file size, the newly defined file area is zero-filled, provided that actual file space is allocated. A new file size can be specified that is beyond the range of current allocated file space, or that is less than current allocated file space.

See Also

DFSExpandFile (page 89), DFSReturnFileMappingInformation (page 97), DFSWrite (page 110)

DFSsopen

Opens the requested file in Direct File Mode

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x, 5.x, 6.x

Platform: NLM

Service: Direct File System

Syntax

```
#include <nwdfs.h>

LONG DFSsopen  (
   BYTE   *fileName,
   LONG    access,
   LONG    share,
   LONG    permission,
   LONG    flagBits,
   LONG    dataStream);
```

Parameters

fileName

(IN) Points to the name of the file to be opened. The file name must be NULL-terminated and must include the path including the volume name but not the server name.

access

(IN) Specifies the access mode.

share

(IN) Specifies the sharing mode of the file.

permission

(IN) Specifies the access permissions for the file. The access permissions (see the sys\stat.h file) are as follows:

S_IWRITE	The file is writable
S_IREAD	The file is readable

flagBits

(IN) Specifies the following flags when a file is opened:

0x00000040	FILE_WRITE_THROUGH_BIT
0x00010000	NO_RIGHTS_CHECK_ON_OPEN_BIT

dataStream

(IN) Specifies the name space.

Return Values

!= -1	The requested operation is complete. The actual value returned is a file handle which is used for other functions that operate on the file.
== -1	The requested file was not opened.

If an error has occurred, `errno` can be set to

1	No such file
4	Bad file handle
6	Permission denied
9	Invalid argument

When an error occurs, `NetWareErrno` is set to

Decimal	Hex	Constant
108	(0x6C)	ERR_BAD_ACCESS
152	(0x98)	ERR_INVALID_VOLUME
156	(0x9C)	ERR_INVALID_PATH

Remarks

The name of the file to be opened is given by the `filename` parameter. The file is accessed according to the access mode specified by the `access` parameter.

When a file is opened in direct file mode by calling DFSsopen, DFS flags the file as being in direct file mode. In this mode, the cache and TTS are bypassed for future accesses to the file. Existing cache entries for the file are flushed and a turbo FAT for the file is built if one does not currently exist.

This could cause problems with other applications that have already opened the file in normal non-direct mode. In this case, the file is switched to direct mode, and the program with the file open in normal mode is able to read the file but cannot write to it. A close must be issued for each handle obtained by an open for the file before the file can be reopened for full normal mode access again (see "Input and Output" on page 75).

The `access` parameter can have the following values as defined in FCNTL.H:

Hex	Constant	Description
0x0000	O_RDONLY	open for read only

Hex	Constant	Description
0x0001	O_WRONLY	open for write only
0x0002	O_RDWR	open for read and write
0x0010	O_APPEND	writes done at end of file
0x0020	O_CREAT	create new file
0x0040	O_TRUNC	truncate existing file
0x0080	O_EXCL	exclusive open

The `share` parameter can have the following values as defined in NWSHARE.H:

SH_COMPAT	Sets compatibility mode
SH_DENYRW	Prevents read or write access to the file
SH_DENYWR	Prevents write access of the file
SH_DENYRD	Prevents read access of the file
SH_DENYNO	Permits both read and write access to the file

The `dataStream` parameter can have the following values as defined in nwfile.h:

0 DOS
1 MACINTOSH
2 NFS
3 FTAM
4 LONG
5 NT

See Also

DFSWrite

Writes sectors into a file using DFS (sleeps until completion)

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x, 5.x, 6.x

Platform: NLM

Service: Direct File System

Syntax

```
#include <nwdfs.h>

LONG DFSWrite  (
   LONG    fileHandle,
   LONG    startingSector,
   LONG    sectorCount,
   BYTE   *buffer);
```

Parameters

fileHandle

(IN) Specifies the file handle returned from a prior DFSsopen or DFScreat call for the indicated file.

startingSector

(IN) Specifies the starting sector number in the file (logical offset from beginning of the file) where the write operation is to begin.

sectorCount

(IN) Specifies the number of sectors to be written from the buffer.

buffer

(IN) Points to a contiguous buffer area large enough to contain the number of sectors to be written.

Return Values

Decimal	Constant	Description
0	DFSNormalCompletion	The operation was completed as specified.
4	DFSVolumeSegmentDeactivated	The volume segments on which the file is located have been deactivated by the operating system.

Decimal	Constant	Description
17	DFSHoleInFileError	An attempt was made to write to a file block address for which space has not been allocated (a hole in a sparse file). Space must be allocated by calling DFSExpandFile to fill holes in sparse files before the associated file block address can be written to successfully when using DFS.
18	DFSParameterError	The function caller supplied an invalid parameter.
23	DFSNotInDirectFileMode	A function requiring a file to be opened in direct mode (using DFSsopen or DFScreat) was requested but the file is not open in direct mode.
131	DFSHardIOError	
136	DFSInvalidFileHandle	(A DFS call was made using a file handle which is not valid. Typically an open was omitted or the user inadvertently closed the file before attempting this access).
148	DFSNoWritePrivilege	The current permissions that the file has been opened with do not allow writes to this file.
149	DFSFileDetached	
150	DFSInsufficientMemory	The Direct File System could not obtain memory necessary to complete the requested function.
162	DFSIOLockError	

Remarks

This function performs a write of one or more sectors using a logical zero-based sector offset into the indicated file. Since this function is blocking, control is returned to the caller after all writes relating to the requested write function are completed. If a status indicating a hole was detected during the requested read operation, the operation failed. It is not possible to write beyond the end of the allocated area of a file, or in holes where no blocks are allocated. This function sets a new file size if a write is issued to allocated file space beyond the current file size.

See Also

DFSWriteNoWait

Writes sectors into a file using DFS (returns immediately after initiation)

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x, 5.x, 6.x

Platform: NLM

Service: Direct File System

Syntax

```
#include <nwdfs.h>

LONG DFSWriteNoWait  (
   LONG            fileHandle,
   LONG            startingSector,
   LONG            sectorCount,
   BYTE          *buffer,
   struct DFSCallBackParameters   *callBackNode);
```

Parameters

fileHandle

(IN) Specifies the file handle returned from a prior DFSsopen call for the file.

startingSector

(IN) Specifies the starting sector number in the file (logical offset from beginning of file) where the write operation is to begin.

sectorCount

(IN) Specifies the number of sectors to be written from the buffer.

buffer

(IN) Points to a contiguous buffer area large enough to contain the number of sectors to be written.

callBackNode

(IN) Points to a structure used to signal completion of all requested writes for a particular call to DFSWriteNoWait.

Return Values

0	Write operation initiated
-1	Bad file handle

This function can also return the return status codes found in the DFSWrite information above.

Remarks

Operation is identical to DFSWrite except that the current thread of execution is not blocked until the completion of the requested operation (DFSWrite calls DFSWriteNoWait, then waits for the completion to be signalled).

The `localSemaphoreHandle` field contains a local semaphore handle obtained by calling OpenLocalSemaphore. WaitOnLocalSemaphore or ExamineLocalSemaphore should be called to determine when the semaphore has been signalled.

The `completionCode` field contains a zero (or the value already in the field if it has not been zeroed out before DFSWriteNoWait is called) if a bad file handle is passed, and a -1 for all other completions.

See Also

Direct File System Structures

8

This documentation alphabetically lists the direct file system structures and describes their purpose, syntax, and fields.

DFSCallBackParameters

Is used to signal completion of DFSReadNoWait

Service: Direct File System

Defined In: nwdfs.h

Structure

```
struct DFSCallBackParameters
{
   LONG    localSemaphoreHandle ;
   LONG    completionCode ;
};
```

Fields

localSemaphoreHandle

   Specifies a local semaphore handle obtained by calling OpenLocalSemaphore.

completionCode

   Specifies the completion code for DFSReadNoWait.

Remarks

See DFSReadNoWait (page 95) for more information.

FileMapStructure

Service: Direct File System

Defined In: nwdfs.h

Structure

```
struct FileMapStructure
{
   LONG   fileBlock ;
   LONG   volumeBlock ;
   LONG   numberOfBlocks ;
};
```

Fields

`fileBlock`

Specifies the starting logical block (zero-based) of the file for this extent or group of contiguous blocks.

`volumeBlock`

Specifies the actual starting logical volume block (zero-based) of the contiguous volume blocks assigned to the logical file blocks above.

`numberOfBlocks`

Specifies the number of contiguous volume blocks that compose this extent (extent block length).

VolumeInformationStructure

Contains information about a NetWare volume

Service: Direct File System

Defined In: nwdfs.h

Structure

```
struct VolumeInformationStructure
{
   LONG   VolumeAllocationUnitSizeInBytes ;
   LONG   VolumeSizeInAllocationUnits ;
   LONG   VolumeSectorSize ;
   LONG   AllocationUnitsUsed ;
   LONG   AllocationUnitsFreelyAvailable ;
   LONG   AllocationUnitsInDeletedFilesNotAvailable ;
   LONG   AllocationUnitsInAvailableDeletedFiles ;
   LONG   NumberOfPhysicalSegmentsInVolume ;
   LONG   PhysicalSegmentSizeInAllocationUnits [64];
};
```

Fields

VolumeAllocationUnitSizeInBytes

Specifies the number of bytes contained in a block allocated by the OS (this can be 4K, 8K, 16K, 32K, or 64K).

VolumeSizeInAllocationUnits

Specifies the number of blocks of the size indicated in the parameter above that are contained in a volume (the volume total size can be calculated with these two parameters).

VolumeSectorSize

Specifies the size of each sector on a volume (currently only a sector size of 512 bytes is supported by the OS).

AllocationUnitsUsed

Specifies the number of blocks on a volume used with current non-deleted files.

AllocationUnitsFreelyAvailable

Specifies the number of blocks currently available for file allocation.

AllocationUnitsInDeletedFilesNotAvailable

Specifies the number of blocks on a volume which compose files that have been deleted but for which the necessary time has not yet elapsed before they can be purged or moved to the `AllocationUnitsFreelyAvailable` category.

AllocationUnitsInAvailableDeletedFiles

Specifies the number of blocks which compose files deleted for which the required time has expired prior to being purged, but which have not yet been purged or moved to the AllocationUnitsFreelyAvailable category.

NumberOfPhysicalSegmentsInVolume

Specifies the number of physical volume segments that are linked to form a volume.

PhysicalSegmentSizeInAllocationUnits

Specifies an array that specifies the number of blocks in each volume segment, of which a maximum of 64 are allowed per volume. This also allows an application process to determine at what point in the logical volume block number a transition takes place from one volume segment to another. This information is needed by applications doing specific file allocation.

DOS Partition Concepts

9

This documentation describes DOS partition, its functions, and features.

NOTE: DOS Partition Services provide functions for NLM development.

The DOS Partition functions allow developers to access files that are in a disk's DOS partition. These functions should be used only when it is absolutely necessary to access a file in the DOS partition. Accessing files in the DOS partition is much slower than accessing files in the NetWare® partition of a disk, and adversely affect other aspects of the server's performance.

The DOS partition refers to the set of files accessible from the server PC booted under DOS. The set of files includes files on floppy disks, on DOS partitions on hard disks, and on any other DOS drive.

A DOS drive can include RAM disks and network drives. However, accessing RAM disks and network drives from NetWare is not recommended. Both RAM disks and NetWare 3.x and above use extended memory and temporarily switch to protected mode, causing a conflict. Accessing network drives uses the LAN board. Since the NetWare 3.x and above OS, in most configurations, also uses this board, two programs would access the same board. For many kinds of LAN boards, this causes a deadlock. Therefore, NLM applications that use the DOS partition should only reference files on floppy or hard disks.

The DOS partition functions have been made available primarily for the installation of new software from the DOS partition to the NetWare partition. If a file in the DOS partition is to be accessed more than once or twice, it should be moved into the NetWare partition.

Additional functions for accessing the DOS partition are available through the File System Services and Operating System I/O Services. You can call these functions on the DOS partition just as you would for the NetWare partition. The functions available for NetWare 3.x and above are open (page 229), fopen (page 274), and freopen (page 285). The same functions are available for NetWare 4.x, 5.x, and 6.x along with these additional functions: access, chmod, remove, and rename (Multiple and Inter-File Management).

The DOS partition functions have been made available primarily for the installation of new software from the DOS partition to the NetWare partition. If a file in the DOS partition is to be accessed more than once or twice, it should be moved into the NetWare partition.

DOSPresent (page 133) should be called before using any of the DOS partition functions.

9.1 DOS Partition Functions

These functions allow you to manage DOS partitions:

DOSClose	Closes a file in the DOS partition.
DOSCopy	Copies a file from the DOS partition to the NetWare® partition.
DOSCreate	Creates a file in the DOS partition of the disk.
DOSFindFirstFile	Initializes a search for files in the DOS partition.
DOSFindNextFile	Searches for files in the DOS partition.

DOSOpen	Opens a file in the DOS partition of the disk.
DOSPresent	Determines whether DOS is still present in memory.
DOSRead	Reads from a file in the DOS partition.
DOSSetDateAndTime	Sets the modification date and time for a file on the DOS partition.
DOSsopen	Opens a DOS file for shared access.
DOSWrite	Writes to a file in the DOS partition.

DOS Partition Functions

<div style="text-align: right; font-size: 3em;">10</div>

This documentation alphabetically lists the DOS partition functions and describes their purpose, syntax, parameters, and return values.

DOSChangeFileMode

Changes a file's attributes

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 4.1, 5.x, 6.x

Platform: NLM

Service: DOS Partition

Syntax

```
#include <nwdos.h>

int DOSChangeFileMode (
      const char  *name,
      LONG        *attributes,
      LONG         function,
      LONG         newAttributes);
```

Parameters

name

   (IN) Points to the path, including the file name.

attributes

   (OUT) Points to the returned attributes of the file.

function

   (IN) Specifies the type of attributes that you want to change:

   0 Read attributes
   1 Set attributes

newAttributes

   (IN) Points to the new attributes to set for `name`.

DOSClose

Closes a file in the DOS partition

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 3.x, 4.x, 5.x, 6.x

Platform: NLM

Service: DOS Partition

Syntax

```
#include <nwdos.h>

int DOSClose  (
   int   handle);
```

Parameters

handle

   (IN) Specifies the file handle obtained by DOSCreate or DOSOpen.

Return Values

Decimal	Hex	Constant
0	(0x00)	ESUCCESS
DOSCode		UNSUCCESSFUL

See Also

DOSCreate (page 127), DOSOpen (page 132), DOSPresent (page 133)

DOSCopy

Copies a file from the DOS partition to the NetWare® partition

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 3.x, 4.x, 5.x, 6.x

Platform: NLM

Service: DOS Partition

Syntax

```
#include <nwdos.h>

int DOSCopy  (
   const char  *NetWareFileName,
   const char  *DOSFileName);
```

Parameters

NetWareFileName

(IN) Points to the name of the file in the NetWare partition (full NetWare paths, including volume names, are allowed).

DOSFileName

(IN) Specifies the name of the file to be copied from the DOS partition (any legal DOS path name is allowed).

Return Values

Decimal	Hex	Constant
0	(0x00)	ESUCCESS
DOSCode		UNSUCCESSFUL

Remarks

If a file with the same name already exists in the NetWare partition, the new file overwrites it.

See Also

DOSCreate

Creates a file in the DOS partition

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 3.x, 4.x, 5.x, 6.x

Platform: NLM

Service: DOS Partition

Syntax

```
#include <nwdos.h>

int DOSCreate  (
   const char  *fileName,
   int         *handle);
```

Parameters

fileName

   (IN) Points to the DOS filename of the file to be created (any legal DOS path name is allowed).

handle

   (OUT) Points to a file handle which provides access to the DOS file.

Return Values

Decimal	Hex	Constant
0	(0x00)	ESUCCESS
DOSCode		UNSUCCESSFUL

Remarks

If the file does not exist, DOSCreate creates it with read/write access. If the file does exist, it is truncated to zero bytes in length.

See Also

DOSOpen (page 132), DOSPresent (page 133)

DOSFindFirstFile

Searches for files in the DOS partition

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 3.x, 4.x, 5.x, 6.x

Platform: NLM

Service: DOS Partition

Syntax

```
#include <nwdos.h>

int DOSFindFirstFile  (
   const char     *fileName,
   WORD            searchAttributes,
   struct find_t  *diskTransferAddress);
```

Parameters

fileName

>   (IN) Points to the name of the file to be found in the DOS partition (full paths, including a drive letter and wildcards, are allowed).

searchAttributes

>   (IN) Specifies the type of file for which to search.

diskTransferAddress

>   (OUT) Points to DOS information about the file.

Return Values

Decimal	Hex	Constant
0	(0x00)	ESUCCESS
DOSCode		UNSUCCESSFUL

Remarks

DOSFindFirstFile finds the first file that matches the `fileName` and `searchAttributes` parameters. If wildcards are used in the `fileName` parameter, call DOSFindNextFile to find other files that also match the `fileName` and `searchAttributes` parameters.

`searchAttributes` can have the following values (defined in the nwdos.h file):

0x00 _A_NORMAL Read/Write file

0x01 _A_RDONLY Read only file

0x02 _A_HIDDEN Hidden file

0x04 _A_SYSTEM System file

0x08 _A_VOLID Volume ID entry

0x10 _A_SUBDIR Subdirectory

0x20 _A_ARCH Archive file

See Also

DOSFindNextFile

Searches for files in the DOS partition

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 3.x, 4.x, 5.x, 6.x

Platform: NLM

Service: DOS Partition

Syntax

```
#include <nwdos.h>

int DOSFindNextFile  (
   struct find_t   *diskTransferAddress);
```

Parameters

diskTransferAddress

   (IN/OUT) Points to an address returned by DOSFindFirstFile and receives information about the file.

Return Values

Decimal	Hex	Constant
0	(0x00)	ESUCCESS
DOSCode		UNSUCCESSFUL

Remarks

Consecutive calls to DOSFindNextFile return information about all DOS files which match the `fileName` and `searchAttributes` parameters specified in DOSFindFirstFile.

See Also

DOSMkdir

Creates a directory on the DOS partition of the NetWare server

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 4.x, 5.x, 6.x

Platform: NLM

Service: DOS Partition

Syntax

```
#include <nwdos.h>

int DOSMkdir(
   const char  *dirName);
```

Parameters

dirName

   (IN) Points to the DOS directory name to be created.

Return Values

0	Success
DOS error code	Failure

DOSOpen

Opens a file with read/write access in the DOS partition of the disk

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 3.x, 4.x, 5.x, 6.x

Platform: NLM

Service: DOS Partition

Syntax

```
#include <nwdos.h>

int DOSOpen  (
   const char  *fileName,
   int         *handle);
```

Parameters

fileName

   (IN) Points to the DOS filename of the file to be opened (any legal DOS pathname is allowed).

handle

   (OUT) Points to a file handle which provides access to the DOS file.

Return Values

Decimal	Hex	Constant
0	(0x00)	ESUCCESS
DOSCode		UNSUCCESSFUL

See Also

DOSPresent

Determines whether DOS is still present in memory

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 3.x, 4.x, 5.x, 6.x

Platform: NLM

Service: DOS Partition

Syntax

```
#include <nwdos.h>

int DOSPresent(void);
```

Return Values

Returns a value of 1 if DOS is still present in memory. Otherwise, it returns a value of 0.

Remarks

DOS must be present in memory for any of the other DOS functions to work.

The REMOVE DOS console command is used to remove DOS from memory.

NOTE: In NetWare 6.5 SP1, an installation option allows you to use NetWare as the boot OS instead of DOS. For backwards compatibility, the DOSPresent function treats the NetWare boot OS as if it were DOS and returns 1.

DOSRead

Reads from a file in the DOS partition

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 3.x, 4.x, 5.x, 6.x

Platform: NLM

Service: DOS Partition

Syntax

```
#include <nwdos.h>

int DOSRead  (
   int    handle,
   LONG   fileOffset,
   void  *buffer,
   LONG   numberOfBytesToRead,
   LONG  *numberOfBytesRead);
```

Parameters

handle

   (IN) Specifies the file handle obtained by DOSOpen or DOSCreate.

fileOffset

   (IN) Specifies the position in the file to start reading.

buffer

   (OUT) Points to the data read from the file.

numberOfBytesToRead

   (IN) Specifies the number of bytes to be read from the file.

numberOfBytesRead

   (OUT) Points to the number of bytes actually read from the file.

Return Values

Decimal	Hex	Constant
0	(0x00)	ESUCCESS
DOSCode		UNSUCCESSFUL

See Also

DOSPresent (page 133), DOSWrite (page 144)

DOSRemove

Removes a file from the DOS Partition of the NetWare server

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 4.x, 5.x, 6.x

Platform: NLM

Service: DOS Partition

Syntax

```
#include <nwdos.h>

int DOSRemove  (
   const char  *fileName);
```

Parameters

fileName

   (IN) Points to the DOS file name to be removed.

Return Values

0	Success
DOS error code	Failure

DOSRename

Renames a file

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 4.1, 5.x, 6.x

Platform: NLM

Service: DOS Partition

Syntax

```
#include <nwdos.h>

int DOSRename {
   const char  *srcName;
   const char  *dstName);
```

Parameters

srcName

   (IN) Points to the original path, including the file name.

dstName

   (IN) Points to the new path (and name) for the file.

Return Values

0	Success
DOS error code	Failure

DOSRmdir

Removes a directory from the DOS Partition of the NetWare server

Local Servers: blocking

Remote Servers: blocking

NetWare Servers: 4.x, 5.x, 6.x

Platform: NLM

Service: DOS Partition

Syntax

```
#include <nwdos.h>

int DOSRmdir  (
   const char  *dirName);
```

Parameters

dirName

   (IN) Points to the DOS directory name to be removed.

Return Values

0	Success
DOS error code	Failure

DOSSetDateAndTime

Sets the modification date and time for a file on the DOS partition

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM

Service: DOS Partition

Syntax

```
#include <nwdos.h>

int DOSSetDateAndTime  (
   int    handle,
   LONG   date,
   LONG   time);
```

Parameters

handle

   (IN) Specifies the file that is to be manipulated (returned from DOSOpen).

date

   (IN) Specifies the DOS date to be set.

time

   (IN) Specifies the DOS time to be set.

Return Values

Returns 0 for success or DOS errors for failure.

Remarks

DOSSetDateAndTime sets the last modified date and time on a DOS file. The value for the `handle` parameter is obtained through a call to DOSOpen.

See Also

DOSFindFirstFile (page 128), DOSOpen (page 132), utime (Multiple and Inter-File Services)

DOSShutOffFloppyDrive

Turns off the disk drive light

Local Servers: N/A

Remote Servers: N/A

NetWare Server: 4.1, 5.x, 6.x

Platform: NLM

Service: DOS Partition

Syntax

```
#include <nwdos.h>

void DOSShutOffFloppyDrive (void);
```

Remarks

After you've finished accessing the DOS partion, you can call DOSShutOffFloppyDrive to shut off the drive light and to stop the disk from spinning.

DOSsopen

Opens a DOS file for shared access

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM

Service: DOS Partition

Syntax

```
#include <nwdos.h>

int DOSsopen  (
   const char  *filename,
   int          access,
   int          share,
   int          permission);
```

Parameters

access

   (IN) Specifies the access mode of the file on open (defined in fcntl.h).

share

   (IN) Specifies the sharing mode of the file on open for DOS 3.0+. (0 for DOS 2.x).

permission

   (IN) Specifies a file permissions if file is created:

   0x00 Normal Read/write
   0x01 Read Only

Return Values

Returns a file handle if successful. Otherwise, returns:

-1	The NetWare or DOS error is in `NetWareErrno`. (The DOS and NetWare error codes do not overlap. If `NetWareErrno` = -1, DOS is not present.)

Remarks

DOSsopen is similar to the NetWare partition function sopen except that it does not support O_APPEND and O_BINARY.

The following access modes are defined in fcntl.h:

0x0000 O_RDONLY Read only

0x0001 O_WRONLY Write only

0x0002 O_RDWR Read/Write

The following values can be ORed with the above access modes:

0x0020 O_CREAT Create new file

0x0040 O_TRUNC Truncate existing file

The following share modes are defined in nwshare.h:

0x00 SH_COMPAT Compatibility mode

0x10 SH_DENYRW Deny read/write mode

0x20 SH_DENYWR Deny write mode

0x30 SH_DENYRD Deny read mode

0x40 SH_DENYNO Deny none mode

See Also

DOSUnlink

Removes a file from the DOS Partition of the NetWare server

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 4.x, 5.x, 6.x

Platform: NLM

Service: DOS Partition

Syntax

```
#include <nwdos.h>

int DOSUnlink  (
   const char  *fileName);
```

Parameters

fileName

   (IN) Points to the DOS file name to be removed.

Return Values

0	Success
DOS error code	Failure

DOSWrite

Writes to a file in the DOS partition

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 3.x, 4.x, 5.x, 6.x

Platform: NLM

Service: DOS Partition

Syntax

```
#include <nwdos.h>

int DOSWrite(
   int         handle,
   LONG         fileOffset,
   const void  *buffer,
   LONG         numberOfBytesToWrite,
   LONG        *numberOfBytesWritten);
```

Parameters

handle

   (IN) Specifies the file handle obtained by DOSOpen or DOSCreate.

fileOffset

   (IN) Specifies the position in the file to start writing.

buffer

   (IN) Points to the data to write to the file.

numberOfBytesToWrite

   (IN) Specifies the number of bytes in the buffer to write to the file.

numberOfBytesWritten

   (OUT) Points to the number of bytes written to the file.

Return Values

Decimal	Hex	Constant
0	(0x00)	ESUCCESS
DOSCode		UNSUCCESSFUL

See Also

DOSPresent (page 133), DOSRead (page 134)

DOS Partition Structures

11

This documentation alphabetically lists the DOS partition structures and describes their purpose, syntax, and fields.

find_t

Contains DOS file information

Service: DOS Partition

Defined In: nwdos.h

Structure

```
struct find_t
{
   char            reserved [21];
   char            attrib;
   unsigned short  wr_time;
   unsigned short  wr_date;
   long            size;
   char            name [13];
};
```

Fields

reserved

   Reserved for future use by DOS.

attrib

   Specifies the file's attributes.

wr_time

   Specifies the file's time stamp in DOS format.

wr_date

   Specifies the file's date stamp in DOS format.

size

   Specifies the file's size in bytes.

name

   Specifies the name of the file.

Extended Attribute Concepts

<div style="text-align: right">

12

</div>

This documentation describes Extended Attribute, its functions, and features.

A file's extended attributes are stored as fields in a separate directory entry. This entry is not accessible through conventional means (that is, directory handles and path specifications). Instead, the entry and its fields are referenced by a NetWare® Extended Attribute (NWEA) structure. The structure includes the following:

- Connection ID
- Read/write position
- Extended attribute handle
- Volume number
- Directory entry
- Key used
- Key length
- Key

The information in the NWEA structure is for internal use only. Allocate and maintain space for the structure, but don't modify its values.

12.1 Extended Attribute Functions

Extended Attribute Services include these functions:

NWCloseEA	Closes the specified extended attribute.
NWFindFirstEA	Initializes the process of scanning extended attributes.
NWFindNextEA	Returns NWEA for accessing the next extended attribute.
NWGetEAHandleStruct	Prepares NWEA to be used by NWReadEA (page 174) or NWWriteEA (page 180).
NWOpenEA	Opens the specified extended attribute.
NWReadEA	Reads the next block of data from the specified extended attribute.
NWWriteEA	Writes data to an extended attribute. If the extended attribute doesn't exist, this function attempts to create it.

NOTE: For information about the requirments for using the ...Ext family of extended attribute functions, see UTF-8 Path and Filenames in *Multiple and Inter-File Services*.

Extended Attribute Tasks

13

This documentation describes common tasks associated with Extended Attribute.

13.1 Scanning for Extended Attributes

1 Call NWFindFirstEA (page 158) and NWFindNextEA (page 162) to scan the extended attributes of a file.

NWFindFirstEA (page 158) initializes the scan operation and NWFindNextEA (page 162) returns an NWEA structure for each extended attribute. The returned structure can be passed to NWReadEA (page 174) or NWWriteEA (page 180).

NWFindFirstEA (page 158) takes an NW_IDX structure as input. NW_IDX must identify the file associated with the extended attribute.

2 To prepare NW_IDX, call the Name Space Services function NWGetDirectoryBase.

13.2 Accessing Extended Attributes

Use the following functions to read and write to the extended attributes of a file:

- NWReadEA (page 174)
- NWWriteEA (page 180)

Both functions open the extended attribute before proceeding, and both require a valid NWEA structure as input.

NOTE: There are several ways to prepare this structure before passing it to either function. Refer to the structure information before proceeding.

13.3 Accessing Attribute Selections

To access an extended attribute by name, prepare the NWEA structure by calling NWOpenEA (page 170).

NOTE: NWOpenEA (page 170) doesn't actually open the extended attribute, but fills in the fields in the NWEA structure. The NWOpenEA (page 170) function uses the directory handle/file path of the associated file, the name of the extended attribute, and the name space as arguments.

You can also call the NWGetEAHandleStruct (page 166) function to obtain the NWEA structure if you know the extended attribute name. This function requires a valid NW_IDX structure. Call NWGetDirectoryBase (Multiple and Inter-File Management) . (NWOpenEA (page 170) performs this step for you.)

13.4 Closing Extended Attributes

1 Complete the read or write operation on an extended attribute before closing an extended attribute.

Partial read or write operations aren't allowed. Any data past the end of the last read or write operation is lost when the file is closed.

2 Call NWCloseEA (page 154) to close the extended attribute directory entry after accessing a file's extended attributes.

Extended Attribute Functions

14

This documentation alphabetically lists the extended attribute functions and describes their purpose, syntax, parameters, and return values.

NWCloseEA

Closes the specified Extended Attribute

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT*, Windows* 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Extended Attribute

Syntax

```
#include <nwnamspc.h>
#include <nwea.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY(NWCCODE) NWCloseEA  (
   const NW_EA_HANDLE  N_FAR  *EAHandle);
```

Delphi Syntax

```
uses calwin32

Function NWCloseEA
  (const EAHandle : pNW_EA_HANDLE
) : NWCCODE;
```

Parameters

EAHandle

   (IN) Points to NW_EA_HANDLE.

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL
0x89CF	INVALID_EA_HANDLE
0x89D3	EA_VOLUME_NOT_MOUNTED

Remarks

NWCloseEA must be called to save any changes made to an Extended Attribute. NWCloseEA must be called after a complete Read and/or Write cycle, not after each read or write function. (NWCloseEA should not be called after a find.)

NW_EA_HANDLE is referenced in all Extended Attribute functions. NW_EA_HANDLE is for internal use only; do not manipulate NW_EA_HANDLE in any way.

NCP Calls

0x2222 86 01 Close Extended Attribute Handle

NWCloseEAExt

Closes the specified Extended Attribute, using UFT-8 path and filenames

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 6.5 SP2 or later

Platform: NLM, Windows 2000, Windows XP

Client: 4.90 SP2 or later

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Extended Attribute

Syntax

```
#include <nwnamspc.h>
#include <nwea.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY(NWCCODE) NWCloseEAExt  (
   const NW_EA_HANDLE_EXT  N_FAR  *EAHandle);
```

Parameters

EAHandle

   (IN) Points to NW_EA_HANDLE_EXT.

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL
0x89CF	INVALID_EA_HANDLE
0x89D3	EA_VOLUME_NOT_MOUNTED

Remarks

NWCloseEAExt must be called to save any changes made to an Extended Attribute. NWCloseEAExt must be called after a complete Read and/or Write cycle, not after each read or write function. (NWCloseEAExt should not be called after a find.)

NOTE: NW_EA_HANDLE_EXT is referenced in all Extended Attribute functions. NW_EA_HANDLE_EXT is for internal use only; do not manipulate NW_EA_HANDLE_EXT in any way.

NCP Calls

0x2222 86 01 Close Extended Attribute Handle

NWFindFirstEA

Initializes the find-first/find-next Extended Attribute process

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Extended Attribute

Syntax

```
#include <nwnamspc.h>
#include <nwea.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY(NWCCODE) NWFindFirstEA (
   NWCONN_HANDLE            conn,
   const NW_IDX N_FAR      *idxStruct,
   NW_EA_FF_STRUCT N_FAR   *ffStruct,
   NW_EA_HANDLE N_FAR      *EAHandle,
   pnstr8                   EAName);
```

Delphi Syntax

```
uses calwin32

Function NWFindFirstEA
  (conn : NWCONN_HANDLE;
   Var idxStruct : NW_IDX;
   Var ffStruct : NW_EA_FF_STRUCT;
   Var EAHandle : NW_EA_HANDLE;
   EAName : pnstr8
) : NWCCODE;
```

Parameters

conn

   (IN) Specifies the NetWare® server connection handle.

idxStruct

   (IN) Points to the NW_IDX structure containing the directory entry index.

ffStruct

   (OUT) Points to the NW_EA_FF_STRUCT structure.

EAHandle

(OUT) Points to the NW_EA_HANDLE structure for the Extended Attribute.

EAName

(OUT) Points to the name of the Extended Attribute (optional).

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL
0x0001	No EAs
0x8801	INVALID_CONNECTION
0x8836	INVALID_PARAMETER
0x890A	NLM_INVALID_CONNECTION

Remarks

If any EAs exist for the associated file, NWFindFirstEA returns the NW_EA_HANDLE structure. If no EAs exist, NWFindFirstEA returns 1.

The NW_EA_HANDLE structure can call the NWReadEA and/or NWWriteEA function. Therefore, you do not need to call the NWGetEAHandleStruct function after NWFindFirstEA to initialize a Read or Write.

If you do call the NWGetEAHandleStruct function in preparation for writing, use the `EAName` parameter. When you copy by calling either NWFindFirstEA or the NWFindNextEA function, you must use the `EAName` parameter. If the `EAName` parameter is not needed, it can be passed NULL.

Information for the NW_IDX structure is obtained by calling the NWNSGetMiscInfo or NWGetDirectoryBase function. Functions use the NW_IDX structure to hold information concerning the name space and directory entry index of a file. This is how an application associates an Extended Attribute with a particular directory entry.

NOTE: The NW_EA_HANDLE and NW_EA_FF_STRUCT structures are for internal use only; do not manipulate these structures in any way.

NWFindFirstEA will return INVALID_PARAMETER if NULL is passed to either the `ffStruct` or `EAHandle` parameters.

NCP Calls

0x2222 86 04 Enumerate Extended Attribute

See Also

NWFindNextEA (page 162), NWGetDirectoryBase (Multiple and Inter-File Management), NWGetEAHandleStruct (page 166), NWReadEA (page 174), NWWriteEA (page 180)

NWFindFirstEAExt

Initializes the find-first/find-next Extended Attribute process, using UFT-8 path and filenames

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 6.5 SP2 or later

Platform: NLM, Windows 2000, Windows XP

Client: 4.90 SP2 or later

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Extended Attribute

Syntax

```
#include <nwnamspc.h>
#include <nwea.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY(NWCCODE) NWFindFirstEAExt (
   NWCONN_HANDLE               conn,
   const NW_IDX        N_FAR   *idxStruct,
   NW_EA_FF_STRUCT_EXT N_FAR   *ffStruct,
   NW_EA_HANDLE_EXT    N_FAR   *EAHandle,
   pnstr8                       EAName);
```

Parameters

conn

(IN) Specifies the NetWare® server connection handle.

idxStruct

(IN) Points to the NW_IDX structure containing the directory entry index. See NW_IDX in *Multiple and Inter-File Services*.

ffStruct

(OUT) Points to the NW_EA_FF_STRUCT_EXT structure.

EAHandle

(OUT) Points to the NW_EA_HANDLE_EXT structure for the Extended Attribute.

EAName

(OUT) Points to the name of the Extended Attribute (optional).

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL
0x0001	No EAs
0x8801	INVALID_CONNECTION
0x8836	INVALID_PARAMETER
0x88F0	UTF8_CONVERSION_FAILED
0x890A	NLM_INVALID_CONNECTION

Remarks

If any EAs exist for the associated file, NWFindFirstEAExt returns the NW_EA_HANDLE_EXT structure. If no EAs exist, NWFindFirstEAExt returns 1.

With a NW_EA_HANDLE_EXT structure, you can call the NWReadEAExt and/or NWWriteEAExt function. Therefore, you do not need to call the NWGetEAHandleStructExt function after NWFindFirstEAExt to initialize a Read or Write.

If you do call the NWGetEAHandleStructExt function in preparation for writing, use the `EAName` parameter. When you copy by calling either NWFindFirstEAExt or the NWFindNextEAExt function, you must use the `EAName` parameter. If the `EAName` parameter is not needed, it can be passed NULL.

Information for the NW_IDX structure is obtained by calling the NWGetDirectoryBaseExt function. Functions use the NW_IDX structure to hold information concerning the name space and directory entry index of a file. This is how an application associates an Extended Attribute with a particular directory entry.

NOTE: The NW_EA_HANDLE_EXT and NW_EA_FF_STRUCT_EXT structures are for internal use only; do not manipulate these structures in any way.

NWFindFirstEAExt will return INVALID_PARAMETER if NULL is passed to either the `ffStruct` or `EAHandle` parameters.

NCP Calls

0x2222 86 04 Enumerate Extended Attribute
0x2222 89 54 Enhanced Enumerate Extended Attribute

See Also

NWFindNextEAExt (page 164), NWGetDirectoryBaseExt (Multiple and Inter-File Management), NWGetEAHandleStructExt (page 168), NWReadEAExt (page 177), NWWriteEAExt (page 183)

NWFindNextEA

Returns the NW_EA_HANDLE structure for the next Extended Attribute

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Extended Attribute

Syntax

```
#include <nwnamspc.h>
#include <nwea.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY(NWCCODE) NWFindNextEA (
   NW_EA_FF_STRUCT  N_FAR  *ffStruct,
   NW_EA_HANDLE  N_FAR     *EAHandle,
   pnstr8                   EAName);
```

Delphi Syntax

```
uses calwin32

Function NWFindNextEA
  (Var ffStruct : NW_EA_FF_STRUCT;
   Var EAHandle : NW_EA_HANDLE;
   EAName : pnstr8
) : NWCCODE;
```

Parameters

ffStruct

   (IN/OUT) Points to the NW_EA_FF_STRUCT structure returned by the NWFindFirstEA function.

EAHandle

   (OUT) Points to the NW_EA_HANDLE structure.

EAName

   (OUT) Points to the name of the Extended Attribute (optional).

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL
0x0001	EA_DONE
0x8996	ERR_NO_ALLOC_SPACE
0x89C9	ERR_EA_NOT_FOUND
0x89CF	ERR_INVALID_EA_HANDLE
0x89D1	ERR_EA_ACCESS_DENIED
0x89FB	ERR_UNKNOWN_REQUEST
0x89FF	ERR_BAD_PARAMETER

Remarks

Before calling NWFindNextEA, you must call the NWFindFirstEA function. NWFindNextEA can then be called multiple times until all EAs have been found. EA_DONE is returned when there are no more EAs.

The NW_EA_HANDLE structure can also call the NWReadEA and/or NWWriteEA function. Therefore, do not call the NWGetEAHandleStruct function after the NWFindFirstEA function in order initialize a Read or Write.

If you do call the NWGetEAHandleStruct function in preparation for a Write, use the EAName parameter. When you copy by calling either the NWFindFirstEA or NWFindNextEA function, the EAName parameter must be used. If the EAName parameter is not needed, pass NULL.

The NW_EA_FF_STRUCT structure is used by the NWFindFirstEA function to return a handle to the first or next Extended Attribute.

The NW_EA_HANDLE and NW_EA_FF_STRUCT structures are for internal use only; do not manipulate these structures in any way.

NCP Calls

0x2222 86 04 Enumerate Extended Attribute

See Also

NWFindFirstEA (page 158)

NWFindNextEAExt

Returns the NW_EA_HANDLE_EXT structure for the next Extended Attribute

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 6.5 SP2 or later

Platform: NLM, Windows 2000, Windows XP

Client: 4.90 SP2 or later

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Extended Attribute

Syntax

```
#include <nwnamspc.h>
#include <nwea.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY(NWCCODE) NWFindNextEA (
   NW_EA_FF_STRUCT_EXT  N_FAR  *ffStruct,
   NW_EA_HANDLE_EXT     N_FAR  *EAHandle,
   pnstr8                      EAName);
```

Parameters

ffStruct

(IN/OUT) Points to the NW_EA_FF_STRUCT_EXT structure returned by the NWFindFirstEAExt function.

EAHandle

(OUT) Points to the NW_EA_HANDLE_EXT structure.

EAName

(OUT) Points to the name of the Extended Attribute (optional).

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL
0x0001	EA_DONE
0x88F0	UTF8_CONVERSION_FAILED
0x8996	ERR_NO_ALLOC_SPACE

0x89C9	ERR_EA_NOT_FOUND
0x89CF	ERR_INVALID_EA_HANDLE
0x89D1	ERR_EA_ACCESS_DENIED
0x89FB	ERR_UNKNOWN_REQUEST
0x89FF	ERR_BAD_PARAMETER

Remarks

Before calling NWFindNextEAExt, you must call the NWFindFirstEAExt function. NWFindNextEAExt can then be called multiple times until all EAs have been found. EA_DONE is returned when there are no more EAs.

With a NW_EA_HANDLE_EXT structure, you can also call the NWReadEAExt and/or NWWriteEA function. Therefore, do not call the NWGetEAHandleStructExt function after the NWFindFirstEAExt function in order initialize a Read or Write.

If you do call the NWGetEAHandleStructExt function in preparation for a Write, use the `EAName` parameter. When you copy by calling either the NWFindFirstEAExt or NWFindNextEAExt function, the `EAName` parameter must be used. If the `EAName` parameter is not needed, pass NULL.

The NW_EA_FF_STRUCT_EXT structure is used by the NWFindFirstEAExt and NWFindNextEAExt functions to return a handle to the first or next Extended Attribute.

NOTE: The NW_EA_HANDLE_EXT and NW_EA_FF_STRUCT_EXT structures are for internal use only; do not manipulate these structures in any way.

NCP Calls

0x2222 86 04 Enumerate Extended Attribute
0x2222 89 54 Enhanced Enumerate Extended Attribute

See Also

NWFindFirstEAExt (page 160), NWGetEAHandleStructExt (page 168), NWReadEAExt (page 177), NWWriteEAExt (page 183)

NWGetEAHandleStruct

Fills the NW_EA_HANDLE structure for use in the NWReadEA and NWWriteEA functions

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Extended Attribute

Syntax

```
#include <nwnamspc.h>
#include <nwea.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY(NWCCODE) NWGetEAHandleStruct (
   NWCONN_HANDLE          conn,
   const nstr8 N_FAR    *EAName,
   const NW_IDX N_FAR   *idxStruct,
   NW_EA_HANDLE N_FAR   *EAHandle);
```

Delphi Syntax

```
uses calwin32

Function NWGetEAHandleStruct
  (conn : NWCONN_HANDLE;
   const EAName : pnstr8;
   const idxStruct : pNW_IDX;
   Var EAHandle : NW_EA_HANDLE
) : NWCCODE;
```

Parameters

conn

   (IN) Specifies the NetWare server connection handle.

EAName

   (IN) Points to the string containing the name of the Extended Attribute.

idxStruct

   (IN) Points to the NW_IDX structure containing the directory entry index.

EAHandle

   (IN/OUT) Points to the NW_EA_HANDLE structure containing the handle of the current Extended Attribute.

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x890A	NLM_INVALID_CONNECTION

Remarks

The NW_EA_HANDLE structure is referenced in all Extended Attribute functions. The NWReadEA and NWWriteEA functions use the NW_EA_HANDLE structure to maintain state information. The state information is required to access the Extended Attribute database.

The NW_IDX structure information is obtained by calling the NWNSGetMiscInfo or NWGetDirectoryBase function. Functions use the NW_IDX structure to hold information about the name space and directory entry index of a file. This is how an application associates an Extended Attribute with a particular directory entry.

See Also

NWFindFirstEA (page 158), NWFindNextEA (page 162), NWGetDirectoryBase (Multiple and Inter-File Management), NWReadEA (page 174), NWWriteEA (page 180)

NWGetEAHandleStructExt

Fills the NW_EA_HANDLE_EXT structure for use in the NWReadEAExt and NWWriteEAExt functions

NetWare Server: 6.5 SP2 or later

Platform: NLM, Windows 2000, Windows XP

Client: 4.90 SP2 or later

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Extended Attribute

Syntax

```
#include <nwnamspc.h>
#include <nwea.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY(NWCCODE) NWGetEAHandleStructExt (
   NWCONN_HANDLE          conn,
   const nstr8      N_FAR  *EAName,
   const NW_IDX     N_FAR  *idxStruct,
   NW_EA_HANDLE_EXT N_FAR  *EAHandle);
```

Parameters

conn

   (IN) Specifies the NetWare server connection handle.

EAName

   (IN) Points to the string containing the name of the Extended Attribute.

idxStruct

   (IN) Points to the NW_IDX structure containing the directory entry index. See NW_IDX in *Multiple and Inter-File Services*.

EAHandle

   (IN/OUT) Points to the NW_EA_HANDLE structure containing the handle of the current Extended Attribute.

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL
0x88F0	UTF8_CONVERSION_FAILED

0x89FF	HARDWARE_FAILURE

Remarks

The NW_EA_HANDLE_EXT structure is referenced in all Extended Attribute functions. The NWReadEAExt and NWWriteEAExt functions use the NW_EA_HANDLE_EXT structure to maintain state information. The state information is required to access the Extended Attribute database.

The NW_IDX structure information is obtained by calling the NWGetDirectoryBaseExt function. Functions use the NW_IDX structure to hold information about the name space and directory entry index of a file. This is how an application associates an Extended Attribute with a particular directory entry.

See Also

NWOpenEAExt (page 172), NWFindFirstEAExt (page 160), NWFindNextEAExt (page 164), NWGetDirectoryBaseExt (Multiple and Inter-File Management), NWReadEAExt (page 177), NWWriteEAExt (page 183)

NWOpenEA

Fills the NW_EA_HANDLE structure so it can be used by the NWReadEA and NWWriteEA functions

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Extended Attribute

Syntax

```
#include <nwnamspc.h>
#include <nwea.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY(NWCCODE) NWOpenEA (
   NWCONN_HANDLE        conn,
   NWDIR_HANDLE         dirHandle,
   const nstr8 N_FAR   *path,
   pnstr8               EAName,
   nuint8               nameSpace,
   NW_EA_HANDLE N_FAR  *EAHandle);
```

Delphi Syntax

```
uses calwin32

Function NWOpenEA
  (conn : NWCONN_HANDLE;
   dirHandle : NWDIR_HANDLE;
   const path : pnstr8;
   EAName : pnstr8;
   nameSpace : nuint8;
   Var EAHandle : NW_EA_HANDLE
) : NWCCODE;
```

Parameters

conn

   (IN) Specifies the NetWare server connection handle.

dirHandle

   (IN) Specifies the NetWare directory handle pointing to the directory to search.

path

> (IN) Points to a path.

EAName

> (IN) Points to the string containing the name of the Extended Attribute.

nameSpace

> (IN) Specifies the name space of the Extended Attribute (see Name Space Flag Values (Multiple and Inter-File Services)).

EAHandle

> (IN/OUT) Points to the NW_EA_HANDLE structure containing the handle of the current Extended Attribute.

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x890A	NLM_INVALID_CONNECTION
0x8998	VOLUME_DOES_NOT_EXIST
0x899B	BAD_DIRECTORY_HANDLE
0x899C	INVALID_PATH

Remarks

NWOpenEA combines the functionality of the NWGetDirectoryBase and NWGetEAHandleStruct functions in one function.

The NWFindFirstEA and NWFindNextEA functions also return a filled NW_EA_HANDLE structure.

NCP Calls

0x2222 87 22 Generate Directory Base and Volume Number

See Also

NWFindFirstEA (page 158), NWFindNextEA (page 162), NWGetDirectoryBase (Multiple and Inter-File Management), NWGetEAHandleStruct (page 166), NWReadEA (page 174), NWWriteEA (page 180)

NWOpenEAExt

Fills the NW_EA_HANDLE_EXT structure so it can be used by the NWReadEAExt and NWWriteEAExt functions

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 6.5 SP2 or later

Platform: NLM, Windows 2000, Windows XP

Client: 4.90 SP2 or later

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Extended Attribute

Syntax

```
#include <nwnamspc.h>
#include <nwea.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY(NWCCODE) NWOpenEAExt (
   NWCONN_HANDLE           conn,
   NWDIR_HANDLE            dirHandle,
   const nstr8     N_FAR   *path,
   pnstr8                  EAName,
   nuint8                  nameSpace,
   NW_EA_HANDLE_EXT N_FAR  *EAHandle);
```

Parameters

conn

(IN) Specifies the NetWare server connection handle.

dirHandle

(IN) Specifies the NetWare directory handle pointing to the directory to search.

path

(IN) Points to a path. The characters in the string must be UTF-8.

EAName

(IN) Points to the string containing the name of the Extended Attribute, using UTF-8 characters.

nameSpace

(IN) Specifies the name space of the Extended Attribute (see Name Space Flag Values (Multiple and Inter-File Services)).

EAHandle

(IN/OUT) Points to the NW_EA_HANDLE_EXT structure containing the handle of the current Extended Attribute.

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x88F0	UTF8_CONVERSION_FAILED
0x890A	NLM_INVALID_CONNECTION
0x8998	VOLUME_DOES_NOT_EXIST
0x899B	BAD_DIRECTORY_HANDLE
0x899C	INVALID_PATH

Remarks

NWOpenEAExt combines the functionality of the NWGetDirectoryBaseExt and NWGetEAHandleStructExt functions in one function.

The NWFindFirstEAExt and NWFindNextEAExt functions also return a filled NW_EA_HANDLE_EXT structure.

NOTE: For information about the requirments for using the ...Ext family of extended attribute functions, see UTF-8 Path and Filenames in *Multiple and Inter-File Services*.

NCP Calls

0x2222 87 22 Generate Directory Base and Volume Number
0x2222 89 22 Generate Directory Base and Volume Number

See Also

NWFindFirstEAExt (page 160), NWFindNextEAExt (page 164), NWGetDirectoryBaseExt (Multiple and Inter-File Management), NWGetEAHandleStructExt (page 168), NWReadEAExt (page 177), NWWriteEAExt (page 183)

NWReadEA

Reads the next block of data from the specified Extended Attribute

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Extended Attribute

Syntax

```
#include <nwnamspc.h>
#include <nwea.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY(NWCCODE) NWReadEA  (
   NW_EA_HANDLE  N_FAR   *EAHandle,
   nuint32               bufferSize,
   pnuint8               buffer,
   pnuint32              totalEASize,
   pnuint32              amountRead);
```

Delphi Syntax

```
uses calwin32

Function NWReadEA
  (Var EAHandle : NW_EA_HANDLE;
   bufferSize : nuint32;
   buffer : pnuint8;
   totalEASize : pnuint32;
   amountRead : pnuint32
) : NWCCODE;
```

Parameters

EAHandle

>   (IN/OUT) Points to the NW_EA_HANDLE structure, obtained by calling either the NWGetHandleStruct, NWFindFirstEA, NWFindNextEA, or NWOpenEA function.

bufferSize

>   (IN) Specifies the size of the buffer.

buffer

>   (OUT) Points to a data buffer.

`totalEASize`

(OUT) Points to the size of the Extended Attribute.

`amountRead`

(OUT) Points to the total amount of data read with the call (not cumulative across multiple calls).

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL
0x0001	EA-EOF (SUCCESSFUL EOF READ)
0x8833	INVALID_BUFFER_LENGTH
0x8988	INVALID_FILE_HANDLE
0x898C	NO_MODIFY_PRIVILEGES
0x8996	SERVER_OUT_OF_MEMORY
0x899C	INVALID_PATH
0x89C9	EA_NOT_FOUND
0x89CE	EA_BAD_DIR_NUM
0x89CF	INVALID_EA_HANDLE
0x89D1	EA_ACCESS_DENIED
0x89D3	EA_VOLUME_NOT_MOUNTED
0x89D5	INSPECT_FAILURE

Remarks

The data block to be read is determined from the state information identified by the `EAHandle` parameter.

NWReadEA and the NWWriteEA function can perform multiple actions, such as opening or creating an Extended Attribute and then calling the appropriate function. To properly end NWReadEA, call the NWCloseEA function after the last Read or Write.

Extended Attribute values should always be read or written completely. Extended Attributes are not treated like files when transferring. Therefore, partial Reads or Writes are not allowed.

If 0x0000 is returned, more data can be read from the Extended Attribute by calling NWReadEA again. In this case, `EAHandle` is already positioned correctly for the next subsequent call. If 0x0001 is returned, no more data can be read and the data in the buffer was read successfully. If other error values are returned, the data in the buffer is not considered valid.

IMPORTANT: If an Extended Attribute is not read or written completely, data past the end of the last Read or Write may be lost!

The NW_EA_HANDLE structure is referenced in all Extended Attribute functions. NWReadEA and the NWWriteEA function use the NW_EA_HANDLE structure to maintain state information. The state information is required to access the Extended Attribute database. The NW_EA_HANDLE structure is for internal use only; do not manipulate it in any way.

Before calling NWReadEA initially, you must obtain the EAHandle parameter to access the Extended Attribute database. An application can obtain an Extended Attribute handle by calling one of the following functions:

NWFindFirstEA

NWFindNextEA

NWGetEAHandleStruct

NWOpenEA

NWReadEA can be called multiple times until the bytes of data read is equal to the value identified by the totalEASize parameter.

NOTE: The value referenced by the amountRead parameter does not reflect the total number of bytes in the Extended Attribute.

For Reads, the bufferSize parameter must be at least 512 bytes; it can be greater than 512 bytes—but must be in multiples of 512. If the bufferSize parameter is less than 512 bytes, NWReadEA returns INVALID_BUFFER_LENGTH.

The NWEARead function reads up to the number of bytes specified by the bufferSize parameter or until the end of the Extended Attribute data, whichever comes first.

NCP Calls

0x2222 86 03 Read Extended Attribute

See Also

NWReadEAExt

Reads the next block of data from the specified Extended Attribute, using UFT-8 path and filenames

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 6.5 SP2 or later

Platform: NLM, Windows 2000, Windows XP

Client: 4.90 SP2 or later

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Extended Attribute

Syntax

```
#include <nwnamspc.h>
#include <nwea.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY(NWCCODE) NWReadEAExt  (
   NW_EA_HANDLE_EXT  N_FAR  *EAHandle,
   nuint32                  bufferSize,
   pnuint8                  buffer,
   pnuint32                 totalEASize,
   pnuint32                 amountRead);
```

Parameters

EAHandle

(IN/OUT) Points to the NW_EA_HANDLE_EXT structure, obtained by calling either the NWGetHandleStructExt, NWFindFirstEAExt, NWFindNextEAExt, or NWOpenEAExt function.

bufferSize

(IN) Specifies the size of the buffer.

buffer

(OUT) Points to a data buffer.

totalEASize

(OUT) Points to the size of the Extended Attribute.

amountRead

(OUT) Points to the total amount of data read with the call (not cumulative across multiple calls).

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL
0x0001	EA-EOF (SUCCESSFUL EOF READ)
0x8833	INVALID_BUFFER_LENGTH
0x88F0	UTF8_CONVERSION_FAILED
0x8988	INVALID_FILE_HANDLE
0x898C	NO_MODIFY_PRIVILEGES
0x8996	SERVER_OUT_OF_MEMORY
0x899C	INVALID_PATH
0x89C9	EA_NOT_FOUND
0x89CE	EA_BAD_DIR_NUM
0x89CF	INVALID_EA_HANDLE
0x89D1	EA_ACCESS_DENIED
0x89D3	EA_VOLUME_NOT_MOUNTED
0x89D5	INSPECT_FAILURE

Remarks

The data block to be read is determined from the state information identified by the `EAHandle` parameter.

NWReadEAExt and the NWWriteEAExt function can perform multiple actions, such as opening or creating an Extended Attribute and then calling the appropriate function. To properly end NWReadEAExt, call the NWCloseEAExt function after the last Read or Write.

Extended Attribute values should always be read or written completely. Extended Attributes are not treated like files when transferring. Therefore, partial Reads or Writes are not allowed.

If 0x0000 is returned, more data can be read from the Extended Attribute by calling NWReadEAExt again. In this case, `EAHandle` is already positioned correctly for the next subsequent call. If 0x0001 is returned, no more data can be read and the data in the buffer was read successfully. If other error values are returned, the data in the buffer is not considered valid.

IMPORTANT: If an Extended Attribute is not read or written completely, data past the end of the last Read or Write may be lost!

The NW_EA_HANDLE_EXT structure is referenced in all Extended Attribute functions. NWReadEAExt and the NWWriteEAExt function use the NW_EA_HANDLE_EXT structure to maintain state information. The state information is required to access the Extended Attribute database. The NW_EA_HANDLE_EXT structure is for internal use only; do not manipulate it in any way.

Before calling NWReadEAExt initially, you must obtain the `EAHandle` parameter to access the Extended Attribute database. An application can obtain an Extended Attribute handle by calling one of the following functions:

NWFindFirstEAExt
NWFindNextEAExt
NWGetEAHandleStructExt
NWOpenEAExt

NWReadEAExt can be called multiple times until the bytes of data read is equal to the value identified by the `totalEASize` parameter.

NOTE: The value referenced by the `amountRead` parameter does not reflect the total number of bytes in the Extended Attribute.

For Reads, the `bufferSize` parameter must be at least 512 bytes; it can be greater than 512 bytes—but must be in multiples of 512. If the `bufferSize` parameter is less than 512 bytes, NWReadEAExt returns INVALID_BUFFER_LENGTH.

The NWEAReadExt function reads up to the number of bytes specified by the `bufferSize` parameter or until the end of the Extended Attribute data, whichever comes first.

NCP Calls

0x2222 86 03 Read Extended Attribute
0x2222 89 53 Enhanced Read Extended Attribute

See Also

NWFindNextEAExt (page 164), NWFindFirstEAExt (page 160), NWOpenEAExt (page 172), NWWriteEAExt (page 183), NWCloseEAExt (page 156)

NWWriteEA

Writes data to an Extended Attribute

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Extended Attribute

Syntax

```
#include <nwnamspc.h>
#include <nwea.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY(NWCCODE) NWWriteEA (
   NW_EA_HANDLE N_FAR  *EAHandle,
   nuint32              totalWriteSize,
   nuint32              bufferSize,
   const nstr8 N_FAR   *buffer,
   pnuint32             amountWritten);
```

Delphi Syntax

```
uses calwin32

Function NWWriteEA
  (Var EAHandle : NW_EA_HANDLE;
   totalWriteSize : nuint32;
   bufferSize : nuint32;
   buffer : pnuint8;
   amountWritten : pnuint32
) : NWCCODE;
```

Parameters

EAHandle

(IN/OUT) Points to the NW_EA_HANDLE structure returned by the NWGetEAHandleStruct, NWFindFirstEA, NWFindNextEA, or NWOpenEA function.

totalWriteSize

(IN) Specifies the size of the total Write.

bufferSize

(IN) Specifies the size of the buffer.

buffer

(IN) Points to a data buffer.

amountWritten

(OUT) Points to the amount of data written by NWWriteEA (not cumulative across multiple calls).

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL; valid data remains in the Extended Attribute
0x0001	SUCCESSFUL; valid data remains in the buffer, not the Extended Attribute
0x8901	ERR_INSUFFICIENT_SPACE
0x898C	NO_MODIFY_PRIVILEGES
0x899C	INVALID_PATH
0x89C8	MISSING_EA_KEY
0x89C9	EA_NOT_FOUND
0x89CB	EA_NO_KEY_NO_DATA
0x89CE	EA_BAD_DIR_NUM
0x89CF	INVALID_EA_HANDLE
0x89D0	EA_POSITION_OUT_OF_RANGE
0x89D1	EA_ACCESS_DENIED
0x89D2	DATA_PAGE_ODD_SIZE
0x89D3	EA_VOLUME_NOT_MOUNTED
0x89D4	BAD_PAGE_BOUNDARY
0x89FF	HARDWARE_FAILURE

Remarks

If the Extended Attribute does not exist, NWWriteEA attempts to create it.

NWWriteEA returns 0x0000 when there is more data in the Extended Attribute and NWWriteEA needs to be called again. NWWriteEA returns 0x0001 when there is valid data in the buffer but none left in the Extended Attribute.

The NWReadEA function and NWWriteEA can perform multiple actions, such as opening or creating an Extended Attribute and then performing the appropriate function. To properly end NWWriteEA, the NWCloseEA function must be called after the last Read, Write, and/or Find Extended Attribute function.

Extended Attribute values should always be read or written completely. Extended Attributes are not treated like files when transferring. Therefore, partial Reads or Writes are not allowed.

IMPORTANT: If an Extended Attribute is not read or written completely, data past the end of the last Read or Write may be lost!

Before calling NWWriteEA, an application must properly initialize the NW_EA_HANDLE structure to access the Extended Attribute database. An application can initialize the NW_EA_HANDLE structure by calling the NWFindFirstEA, NWFindNextEA, NWGetEAHandleStruct, or NWOpenEA function. The NW_EA_HANDLE structure is for internal use only; do not manipulate it in any way.

For Writes, the `bufferSize` parameter should be at least 512 bytes. If the `bufferSize` parameter is less than the `totalWriteSize` parameter, it must be a multiple of 512.

NWWriteEA writes up to the number of bytes specified by the `bufferSize` parameter or until the end of the Extended Attribute data, whichever comes first. If the data to be written is larger than the buffer size, NWWriteEA must be called multiple times to write all the data to the Extended Attribute.

An application should complete the entire Write before closing the Extended Attribute.

NCP Calls

0x2222 86 02 Write Extended Attribute

See Also

NWWriteEAExt

Writes data to an Extended Attribute, using UFT-8 path and filenames

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 6.5 SP2 or later

Platform: NLM, Windows 2000, Windows XP

Client: 4.90 SP2 or later

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Extended Attribute

Syntax

```
#include <nwnamspc.h>
#include <nwea.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY(NWCCODE) NWWriteEA (
   NW_EA_HANDLE_EXT N_FAR  *EAHandle,
   nuint32                  totalWriteSize,
   nuint32                  bufferSize,
   const nstr8      N_FAR  *buffer,
   pnuint32                 amountWritten);
```

Parameters

EAHandle

(IN/OUT) Points to the NW_EA_HANDLE_EXT structure returned by the NWGetEAHandleStructExt, NWFindFirstEAExt, NWFindNextEAExt, or NWOpenEAExt function.

totalWriteSize

(IN) Specifies the size of the total Write.

bufferSize

(IN) Specifies the size of the buffer.

buffer

(IN) Points to a data buffer.

amountWritten

(OUT) Points to the amount of data written by NWWriteEAExt (not cumulative across multiple calls).

Return Values

These are common return values; see Return Values (*Return Values for C*) for more information.

0x0000	SUCCESSFUL; valid data remains in the Extended Attribute
0x0001	SUCCESSFUL; valid data remains in the buffer, not the Extended Attribute
0x88F0	UTF8_CONVERSION_FAILED
0x8901	ERR_INSUFFICIENT_SPACE
0x898C	NO_MODIFY_PRIVILEGES
0x899C	INVALID_PATH
0x89C8	MISSING_EA_KEY
0x89C9	EA_NOT_FOUND
0x89CB	EA_NO_KEY_NO_DATA
0x89CE	EA_BAD_DIR_NUM
0x89CF	INVALID_EA_HANDLE
0x89D0	EA_POSITION_OUT_OF_RANGE
0x89D1	EA_ACCESS_DENIED
0x89D2	DATA_PAGE_ODD_SIZE
0x89D3	EA_VOLUME_NOT_MOUNTED
0x89D4	BAD_PAGE_BOUNDARY
0x89FF	HARDWARE_FAILURE

Remarks

If the Extended Attribute does not exist, NWWriteEAExt attempts to create it.

NWWriteEAExt returns 0x0000 when there is more data in the Extended Attribute and NWWriteEAExt needs to be called again. NWWriteEAExt returns 0x0001 when there is valid data in the buffer but none left in the Extended Attribute.

The NWReadEAExt function and NWWriteEAExt can perform multiple actions, such as opening or creating an Extended Attribute and then performing the appropriate function. To properly end NWWriteEAExt, the NWCloseEAExt function must be called after the last Read, Write, and/or Find Extended Attribute function.

Extended Attribute values should always be read or written completely. Extended Attributes are not treated like files when transferring. Therefore, partial Reads or Writes are not allowed.

IMPORTANT: If an Extended Attribute is not read or written completely, data past the end of the last Read or Write may be lost!

Before calling NWWriteEAExt, an application must properly initialize the NW_EA_HANDLE_EXT structure to access the Extended Attribute database. An application can

initialize the NW_EA_HANDLE_EXT structure by calling the NWFindFirstEAExt, NWFindNextEAExt, NWGetEAHandleStructExt, or NWOpenEAExt function. The NW_EA_HANDLE_EXT structure is for internal use only; do not manipulate it in any way.

For Writes, the `bufferSize` parameter should be at least 512 bytes. If the `bufferSize` parameter is less than the `totalWriteSize` parameter, it must be a multiple of 512.

NWWriteEAExt writes up to the number of bytes specified by the `bufferSize` parameter or until the end of the Extended Attribute data, whichever comes first. If the data to be written is larger than the buffer size, NWWriteEAExt must be called multiple times to write all the data to the Extended Attribute.

An application should complete the entire Write before closing the Extended Attribute.

NCP Calls

0x2222 86 02 Write Extended Attribute
0x2222 89 52 Enhanced Write Extended Attribute

See Also

NWFindNextEAExt (page 164), NWFindFirstEAExt (page 160), NWOpenEAExt (page 172), NWReadEAExt (page 177), NWCloseEAExt (page 156)

Extended Attribute Structures

15

This documentation alphabetically lists the extended attributes structures and describes their purpose, syntax, and fields.

NW_EA_FF_STRUCT

Maintains state information when scanning an extended attribute file

Service: Extended Attribute

Defined In: nwea.h

Structure

```
typedef struct
{
   NWCONN_HANDLE   connID ;
   nuint16         nextKeyOffset ;
   nuint16         nextKey ;
   nuint32         numKeysRead ;
   nuint32         totalKeys ;
   nuint32         EAHandle ;
   nuint16         sequence ;
   nuint16         numKeysInBuffer ;
   nuint8          enumBuffer [512];
} NW_EA_FF_STRUCT;
```

Delphi Structure

```
uses calwin32

NW_EA_FF_STRUCT = packed Record
    connID : NWCONN_HANDLE;
    nextKeyOffset : nuint16;
    nextKey : nuint16;
    numKeysRead : nuint32;
    totalKeys : nuint32;
    EAHandle : nuint32;
    sequence : nuint16;
    numKeysInBuffer : nuint16;
    enumBuffer : Array[0..511] Of nuint8
  End;
```

Fields

connID

   Specifies a connection to the server storing the Extended Attribute.

nextKeyOffset

   Specifies a value that the server uses as part of an internal handle.

nextKey

   Specifies a value that the server uses as part of an internal handle.

numKeysRead

Specifies the number of keys that have been read from the extended attribute file.

`totalKeys`

Specifies the total number of keys in the extended attribute file.

`EAHandle`

Specifies the handle for the current key.

`sequence`

Specifies the current key number.

`numKeysInBuffer`

Specifies the number of keys in the current buffer.

`enumBuffer`

Specifies the current buffer containing keys read from the extended attribute file.

Remarks

NW_EA_FF_STRUCT is an internal handle for library use only. Applications must not modify this structure in any way.

NW_EA_FF_STRUCT_EXT

Maintains state information when scanning an extended attribute file

Service: Extended Attribute

Defined In: nwea.h

Structure

```
typedef struct
{
   NWCONN_HANDLE   connID ;
   nuint16         nextKeyOffset ;
   nuint16         nextKey ;
   nuint32         numKeysRead ;
   nuint32         totalKeys ;
   nuint32         EAHandle ;
   nuint16         sequence ;
   nuint16         numKeysInBuffer ;
   nuint8          enumBuffer [1530];
} NW_EA_FF_STRUCT_EXT;
```

Fields

connID

>   Specifies a connection to the server storing the Extended Attribute.

nextKeyOffset

>   Specifies a value that the server uses as part of an internal handle.

nextKey

>   Specifies a value that the server uses as part of an internal handle.

numKeysRead

>   Specifies the number of keys that have been read from the extended attribute file.

totalKeys

>   Specifies the total number of keys in the extended attribute file.

EAHandle

>   Specifies the handle for the current key.

sequence

>   Specifies the current key number.

numKeysInBuffer

>   Specifies the number of keys in the current buffer.

enumBuffer

>   Specifies the current buffer containing keys read from the extended attribute file.

Remarks

NW_EA_FF_STRUCT_EXT is an internal handle for library use only. Applications must not modify this structure in any way.

NW_EA_HANDLE

Defines information associated with the extended attribute handle

Service: Extended Attribute

Defined In: nwea.h

Structure

```
typedef struct
{
   NWCONN_HANDLE    connID ;
   nuint32          rwPosition ;
   nuint32          EAHandle ;
   nuint32          volNumber ;
   nuint32          dirBase ;
   nuint8           keyUsed ;
   nuint16          keyLength ;
   nuint8           key [256];
} NW_EA_HANDLE;
```

Delphi Structure

```
uses calwin32

NW_EA_HANDLE = packed Record
    connID : NWCONN_HANDLE;
    rwPosition : nuint32;
    EAHandle : nuint32;
    volNumber : nuint32;
    dirBase : nuint32;
    keyUsed : nuint8;
    keyLength : nuint16;
    key : Array[0..255] Of nuint8
  End;
```

Fields

connID

Specifies the server storing the Extended Attribute.

rwPosition

Specifies the current position within the Extended Attribute file.

EAHandle

Specifies the handle to the Extended Attribute file.

volNumber

Specifies the volume storing the Extended Attribute file.

dirBase

Specifies the directory base associated with the Extended Attribute file.

keyUsed

Specifies the key used to access the Extended Attribute.

keyLength

Specifies the length of the `key` parameter.

key

Specifies the Extended Attribute key.

Remarks

NW_EA_HANDLE is an internal handle for library use only. Applications must not modify this structure in any way.

NW_EA_HANDLE_EXT

Defines information associated with the extended attribute handle

Service: Extended Attribute

Defined In: nwea.h

Structure

```
typedef struct
{
   NWCONN_HANDLE   connID ;
   nuint32         rwPosition ;
   nuint32         EAHandle ;
   nuint32         volNumber ;
   nuint32         dirBase ;
   nuint8          keyUsed ;
   nuint16         keyLength ;
   nuint8          key [766];
} NW_EA_HANDLE_EXT;
```

Fields

connID

> Specifies the server storing the Extended Attribute.

rwPosition

> Specifies the current position within the Extended Attribute file.

EAHandle

> Specifies the handle to the Extended Attribute file.

volNumber

> Specifies the volume storing the Extended Attribute file.

dirBase

> Specifies the directory base associated with the Extended Attribute file.

keyUsed

> Specifies the key used to access the Extended Attribute.

keyLength

> Specifies the length of the key parameter.

key

> Specifies the Extended Attribute key.

Remarks

NW_EA_HANDLE_EXT is an internal handle for library use only. Applications must not modify this structure in any way.

Operating System I/O Concepts

16

This documentation describes Operating System I/O, its functions, and features.

Operating system I/O functions perform nonbuffered I/O operations. The functions in this section reference files using a file handle that is returned when a file is opened. The file handle is passed to the other functions. Files opened at the OS level (with the open (page 229), sopen (page 238), and creat (page 207) functions), or opened at the stream level and referenced with the fileno (page 271) function are called *first-level* open files.

NOTE: As used in this chapter, streams are standard files and are not to be confused with NetWare® STREAMS.

Operating System I/O provides functions for NLM development.

16.1 File Permission Conversion

When a file is created during the operation of creat, open, or sopen, the `permission` parameter can be specified as S_IWRITE (writable), S_IREAD (readable) or S_IWRITE | S_IREAD (writable and readable). 0 also allows both writing and reading.

Files created with the S_IWRITE option can be written to, modified, and deleted by any object having such rights to the file. Files created with S_IREAD are created as read-only. S_IREAD is converted to directory attributes that prohibit writing, renaming, deletion, copying, or the ability to migrate or compress.

16.2 File Paths

When specifying a file to an Operating System I/O function, the file paths include the following conditions:

- The file paths do not have drive letters.

  File path drive letters are not used in NLM™ applications.
- File paths can contain volume names.

  Any volumes mounted on the server may be referenced by the NLM. Volume names are from 2 to 15 characters long.

The syntax for a file path that includes a volume name is as follows:

volume: directory\...\directory\filename

Each thread group in an NLM has its own current working directory (CWD). If a relative path is specified, it is assumed to be relative to the thread group's CWD. When a thread group is started, the initial value of the CWD is "SYS:" (root directory on the SYS volume). The only exception to this is when the (CLIB_OPT) parameters are specified when the NLM is loaded. These parameters change the CWD for the initial thread group.

NOTE: The maximum number of file handles that can be open at once for NetWare® 4.x is 1700.

NLM applications can open a given file more than once. The file handle and task information for subsequent opens depend on the circumstances of the open, as follows:

- If a file is opened more than once by a particular thread, using the same connection and task, then the same file handle is returned. A count of the number of times a file is opened with a particular handle is associated with each handle. A handle remains usable as long as its open count is greater than zero.

- If a file is opened more than once by different threads or by the same thread but with a different connection or task (than a previous connection or task with which the file was opened), then a different handle is returned. Closing one handle for a given file has no effect on any other handles to that file.

- If a file is opened more than once with the same connection and task but with different threads, then the second (and subsequent) open is done with a newly allocated task number. The task number is automatically allocated on the current connection by open (page 229) (or sopen (page 238) or creat (page 207)). The current task number is not affected.

I/O redirection on first-level files and file handles is supported *only* for NetWare 4.x and above (see dup (page 209) and dup2 (page 211)). Redirection of second-level files is supported for all NetWare versions. Second-level files include those opened with fopen (page 274), fdopen (page 256), or freopen (page 285).

Text mode for first-level file handles is *not* supported. Only binary mode is supported. (In binary mode, data is transmitted unchanged. In text mode, carriage-return/line-feed pairs are translated to line feeds on input, and line feeds are translated to carriage-return/line-feed pairs on output.)

The following handles are predefined and always available:

STDIN (0)—Input from the current screen

STDOUT (1)—Output to the current screen

STDERR (2)—Output to the current screen

16.3 Operating System I/O Functions

These are the functions to handle OS input and output:

chsize	Changes the file size.
close	Closes a file, stream, or BSD socket.
creat	Creates and opens a file or stream.
dup	Returns a file handle that refers to the same open file as handle. Supported only for the NetWare® 4.x and above OS.
dup2	Forces file handle handle2 to reference the same open file as handle Supported only for NetWare 4.x and above.
eof	Determines if the end of the file has been reached for a specified file.
fcntl	Provides control over open files.
filelength	Returns the number of bytes in an open file.
fstat	Obtains information about an open file.

isatty	Tests whether the specified handle refers to a screen or not.
lock	Locks a portion of a file.
lseek	Sets the current file position.
open	Opens a file, stream, or socket.
read	Reads data from a file, stream, or socket.
sopen	Opens a file, stream, or socket for shared access.
tell	Determines the current file position.
unlock	Unlocks a previously locked portion of a file.
write	Writes data to a file, stream, or socket.

Operating System I/O Functions

17

This documentation alphabetically lists the operating system I/O functions and describes their purpose, syntax, parameters, and return values.

cancel

Cancels any current call to setvbuf for the specified thread

Local Servers: blocking

Remote Servers: N/A

Classification: 4.x, 5.x, 6.x

Service: Operating System I/O

Syntax

```
#include <unistd.h>

int cancel  (
   int   t_id);
```

Parameters

t_id

   (IN) Specifies the target thread id, exactly as returned when the target thread as created.

Return Values

Returns 0 on success. If an error occurs, it returns -1 and `errno` is set to:

0x0000004E	ECANCELED	The operation is invalid because the target thread is either not in a delay operation or the delay operation has already been cancelled.

Remarks

A CLib thread was optionally awakened from delay by calling ResumeThread (Threads Management) and, to overcome the affects of delay, cancel must be called.

See Also

setvbuf (page 321), GetThreadID (Threads Management)

chsize

Changes the file size

Local Servers: blocking

Remote Servers: blocking

Platform: NLM

Service: Operating System I/O

Syntax

```
#include <unistd.h>

int chsize  (
   int    handle,
   LONG   size);
```

Parameters

handle

   (IN) Specifies a file handle.

size

   (IN) Specifies the file size.

Return Values

chsize returns a value of 0 if successful. It returns a value of -1 if an error occurs.

If an error occurs, `errno` is set to:

Decimal	Constant	Description
4	EBADF	Bad file number.
6	EACCES	Permission denied.
12	ENOSPC	No space left on device.

If chsize does not complete successfully, `NetWareErrno` is set.

Remarks

The chsize function changes the size of the file associated with the file handle. It can truncate or extend the file, depending on the value of `size` compared to the file's original size.

The mode in which the file was opened must allow writing.

If chsize extends the file, it appends NULL characters (\0). If it truncates the file, all data beyond the new end-of-file indicator is lost.

See Also

eof (page 213), filelength (page 216)

close

Closes a file or stream

Local Servers: blocking

Remote Servers: blocking

Classification: POSIX

Platform: NLM

Service: Operating System I/O

Syntax

```
#include <unistd.h>

int close  (
   int   handle);
```

Parameters

handle

   (IN) Specifies a file handle.

Return Values

When an error occurs while closing the file, a value of -1 is returned. Otherwise, a value of 0 is returned.

If an error occurs, `errno` is set to:

4	EBADF	Bad file number.

If the function does not complete successfully, `NetWareErrno` is set.

Remarks

This function also works on the DOS partition.

The `handle` value is the file handle returned by a successful execution of the open, sopen, or creat function. After a file is closed, the file handle is no longer valid and should not be reused.

UNIX STREAMS: If a Stream file is closed and the calling process had previously registered to receive a SIGPOLL signal for events associated with that file, the calling process is unregistered for events associated with the file.

The last close for a Stream causes the Stream associated with `handle` to be dismantled.

- If O_NDELAY is not set and no signals have been posted for the Stream, the close function waits up to 15 seconds for each module and driver and for any output to drain before dismantling the Stream.

- If the O_NDELAY flag is set or if there are any pending signals, the close function does not wait for output to drain and dismantles the Stream immediately.

See Also

creat

Creates and opens a file or stream

Local Servers: blocking

Remote Servers: blocking

Classification: POSIX

Platform: NLM

Service: Operating System I/O

Syntax

```
#include <fcntl.h>

int creat  (
   const char  *filename,
   int          permission);
```

Parameters

filename

   (IN) Points to the name of the file to be created (considered within the context of the currently set name space).

permission

   (IN) Specifies the access permissions for the file.

Return Values

When an error occurs while creating the file, a value of -1 is returned. Otherwise, an integer (not equal to -1), known as the file handle, is returned to be used with the other functions that operate on the file.

When an error occurs, `errno` can be set to:

Decimal	Constant	Description
1	ENONENT	No such file.
6	EACCES	Permission denied.
9	EINVAL	Invalid argument.

When an error occurs, `NetWareErrno` is set to:

Decimal	Hex	Constant	Description
152	(0x98)	ERR_INVALID_VOLUME	

Decimal	Hex	Constant	Description
156	(0x9C)	ERR_INVALID_PATH	
191	(0xBF)	ERR_INVALID_NAME_SPACE	On remote servers when using a non-DOS name space.

Remarks

This function also works on the DOS partition.

This function allows for as many open files as there is available memory. The `filename` parameter supplies the name of the file to be created. If the file exists (the current connection must have Write rights), it is truncated to contain no data and the preceding permission setting is unchanged.

If the file does not exist, it is created with access permission given by the `permission` parameter.

The access permission for the file is specified as a combination of bits (defined in the SYS\STAT.H header file):

S_IWRITE	The file is writable.
S_IREAD	The file is readable.

The `permission` parameter can be specified as S_IWRITE, S_IREAD or S_IWRITE|S_IREAD. Specifying 0 also makes a file both writable and readable.

The current connection must have Create rights to create a new file or have Read/Write rights to write to a file that already exists.

See Also

Example

```
#include   <stddef.h>
#include   <fcntl.h>
#include   <errno.h>
#include   <string.h>

main()
{
   int   fh;
   if((fh = creat("name",0)) == -1)
      printf ("creat() error\n");
   else
      close (fh);
}
```

dup

Returns a file handle that refers to the same open file as handle (supported only for NetWare 4.x and above)

Local Servers: nonblocking

Remote Servers: nonblocking

Classification: POSIX

Platform: NLM

Service: Operating System I/O

Syntax

```
#include <unistd.h>

int dup  (
   int   handle);
```

Parameters

handle

(IN) Specifies the file handle that is to be duplicated.

Return Values

When an error occurs, a value of -1 is returned. Otherwise, the return value is a nonnegative integer that is the file handle.

If an error occurs, `errno` can be set to:

Decimal	Constant	Description
4	EBADF	Bad file number
11	EMFILE	Too many open files

Remarks

The dup function duplicates a file handle by returning a file handle that refers to the same open file as `handle`. Since both handles reference the same file, either handle can be used for operations on the file.

NOTE: For an example of how to reverse the effect of redirecting `stdin`, see the example for .

See Also

dup2

Forces the file handle to reference the same open file as the `handle` parameter (supported only for NetWare 4.x and above)

Local Servers: nonblocking

Remote Servers: nonblocking

Classification: POSIX

Platform: NLM

Service: Operating System I/O

Syntax

```
#include <unistd.h>

int dup2  (
   int   handle,
   int   handle2);
```

Parameters

handle

   (IN) Specifies the file handle to be duplicated.

handle2

   (IN) Specifies the file handle to be forced to reference the same file as the `handle` parameter.

Return Values

When an error occurs, a value of -1 is returned. Otherwise, the return value is a nonnegative integer that is the file handle.

If an error occurs, `errno` can be set to:

Decimal	Constant	Description
4	EBADF	Bad file number
11	EMFILE	Too many open files

Remarks

The `handle` parameter must be a handle to a file that is already open. If `handle2` references a file that is already open, that file is closed before `handle2` is forced to reference the file for `handle`.

See Also

close (page 205), creat (page 207), dup (page 209), eof (page 213), filelength (page 216), fileno (page 271), fstat (page 217), ftell (page 293), isatty (page 223), lseek (page 226), open (page 229), read (page 234), sopen (page 238), write (page 246)

eof

Determines if the end of the file has been reached for a specified file

Local Servers: nonblocking

Remote Servers: nonblocking

Classification: POSIX

Platform: NLM

Service: Operating System I/O

Syntax

```
#include <unistd.h>

int eof  (
   int   handle);
```

Parameters

handle

   (IN) Specifies a file handle.

Return Values

eof returns a value of 1 if the current file position is at the end of the file. If the current file position is not at the end, a value of 0 is returned. If an error is detected, a value of -1 is returned.

If an error occurs, `errno` is set to:

4	EBADF	Bad file number.

If eof does not complete successfully, `NetWareErrno` is set.

Remarks

The eof function determines if the end of the file has been reached for the file whose file handle is given by `handle`. Because the current file position is set following an input operation, eof can be called to detect the end of the file before an input operation beyond the end of the file is attempted.

See Also

dup (page 209), dup2 (page 211), read (page 234)

fcntl

Controls file handles

Local Servers: nonblocking

Remote Servers: nonblocking

Classification: POSIX

Platform: NLM

Service: Operating System I/O

Syntax

```
#include <fcntl.h>

int fcntl  (
   int   handle,
   int   cmd,
   int   arg);
```

Parameters

handle

> (IN) Specifies a file handle to be operated on by `cmd`.

cmd

> (IN) Specifies one of two commands to act on the specified file handle.

arg

> (IN/OUT) Specifies the handle status flags.

Return Values

Upon successful completion of the F_GETFL command, fcntl returns the current value of the requested flag. Otherwise, a value of -1 is returned and `errno` indicates the error.

Decimal	Constant	Description
4	EBADF	The specified file handle is not a valid one.
9	EINVAL	Either `cmd` or `val` is not supported.
35	EWOULDBLOCK	Either no data is available to a read call or a write operation is in a blocking mode.

Remarks

The fcntl function provides for file control over file handles. The `handle` parameter is a file handle to be operated on by `cmd`. The `cmd` parameter includes either the F_GETFL or the F_SETFL commands described below:

F_GETFL	Get handle status flags.
F_SETFL	Set handle status flags.

Flags are passed in the `arg` parameter. The FNDELAY flag is defined for the F_GETFL and F_SETFL commands. It establishes a nonblocking I/O mode; if no data is available to a read call or if a write operation is in a blocking mode, the call returns a value of -1 with the error EWOULDBLOCK.

See Also

ioctl (page 219)

filelength

Returns the number of bytes in an open file

Local Servers: blocking

Remote Servers: blocking

Platform: NLM

Service: Operating System I/O

Syntax

```
#include <nwfileio.h>

LONG filelength  (
   int   fildes);
```

Parameters

handle

> (IN) Specifies a file handle.

Return Values

filelength returns a value of -1 if an error occurs.

If an error occurs, `errno` can be set to:

4	EBADF	Bad file number.

If filelength does not complete successfully, `NetWareErrno` is set.

Remarks

The filelength function returns the number of bytes in the opened file indicated by the file handle.

See Also

Example

```
#include <fcntl.h>

LONG   length;
int    handle;
length = filelength (handle);
```

fstat

Obtains information about an open file

Local Servers: blocking

Remote Servers: blocking

Classification: POSIX

Platform: NLM

Service: Operating System I/O

Syntax

```
#include <sys\stat.h>

int fstat (
   int            handle,
   struct stat    *statblk);
```

Parameters

handle

(IN) Specifies a file handle.

statblk

(OUT) Points to the address of the structure stat.

Return Values

fstat returns a value of 0 when the information is obtained successfully. Otherwise, a value of -1 is returned.

If an error occurs, `errno` is set to:

4	EBADF	Bad file number.

If fstat does not complete successfully, `NetWareErrno` is set.

Remarks

The fstat function obtains information about an open file whose file handle is `handle`. This information is placed in the structure located at the address indicated by the `statblk` parameter.

The SYS\STAT.H header file contains definitions for stat and describes the contents of the fields within that structure.

See Also

Example

```c
#include <stdio.h>
#include <nwtypes.h>
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>

main()
{
   int          handle;
   struct stat  buf;
   handle = open ("test.dat",O_RDONLY | O_BINARY,0);
   if(handle == -1)
   {
      printf ("could not open file");
      exit (0);
   }
   if(fstat (handle,&buf) == -1)
      printf ("fstat error\r\n");
   close(handle);
   printf ("st_dev = %x\r\n",buf.st_dev);
   printf ("st_ino = %x\r\n",buf.st_ino);
   printf ("st_mode = %x\r\n",buf.st_mode);
   printf ("st_nlink = %x\r\n",buf.st_nlink);
   printf ("st_uid = %x\r\n",buf.st_uid);
   printf ("st_gid = %x\r\n",buf.st_gid);
   printf ("st_rdev = %x\r\n",buf.st_rdev);
   printf ("st_size = %x\r\n",buf.st_size);
   printf ("st_atime = %x\r\n",buf.st_atime);
   printf ("st_mtime = %x\r\n",buf.st_mtime);
   printf ("st_ctime = %x\r\n",buf.st_ctime);
   printf ("st_btime = %x\r\n",buf.st_btime);
   printf ("st_attr = %x\r\n",buf.st_attr);
   printf ("st_archivedID = %x\r\n",buf.st_archivedID);
   printf ("st_updatedID = %x\r\n",buf.st_updatedID);
   printf ("st_inheritedRightsMask =
            %x\r\n",buf.st_inheritedRightsMask);
   printf ("st_originatingNameSpace =
            %c\r\n",buf.st_originatingNameSpace);
   printf ("st_name = %s\r\n",buf.st_name);
/*————- new fields starting in v. 4.11 ————-*/
   printf ("st_name2 = %s\r\n",buf.st_name2);
   printf ("st_blksize = %x\r\n",buf.st_blksize);
   printf ("st_blocks = %x\r\n",buf.st_blocks);
   printf ("st_flags = %x\r\n",buf.st_flags);
}
```

ioctl

Performs a variety of control functions on a STREAMS, BSD Socket, and pipe file descriptors

Local Servers: blocking

Remote Servers: N/A

Classification: UNIX (nonstandard)

NetWare Server: 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM

Service: Operating System I/O

Syntax

```
#include <sys/ioctl.h>
#include <stropts.h>

int ioctl  (
   int    filedes,
   int    command,
   void  *arg);
```

Parameters

filedes

   (IN) Specifies a descriptor returned from the open, pipe, sopen, socket, etc., functions.

command

   (IN) Specifies the control function to be performed.

arg

   (IN/OUT) Points to an additional argument to be used or points to an argument returned by ioctl (depending on the control function performed).

Return Values

Upon successful completion, the value returned depends upon the control function (command argument) but must be a nonnegative integer. Otherwise, `errno` indicates the occurring error.

Remarks

See spxc_rw.c (NDK: NetWare Server Protocol Libraries for C)(unsupported) for sample code.

For Stream files, ioctl performs the following control operations:

FIOGETNBIO	Supported for TCP, UDP, ICMP, RAWIP sockets. Returns the nonblocking status of the socket. The status is returned in the third parameter of ioctl as a value and result parameter. A nonzero value indicates the flag is SET, and a zero value indicates the flag is CLEAR.
FIONBIO	Supported for TCP, UDP, ICMP, RAWIP sockets. Sets and clears the nonblocking flag for the sockets, depending on the value of the third parameter that is passed to ioctl. A nonzero value indicates the flag is SET, and a zero value indicates the flag is CLEAR.
FIONREAD	Supported for TCP, UDP, ICMP, RAWIP sockets. Returns the number of bytes that are currently in the socket receive buffer. The value is returned in the third parameter of ioctl.
I_FDINSERT	Creates a message from user-specified buffers, adds information about another Stream and sends the message downstream. The message contains a control part and an optional data part. On failure, `errno` is set.
I_FIND	Compares the names of all modules currently present in the Stream to the name pointed to by the `arg` parameter, and returns a value of 1 if the named module is present in the Stream. It returns a value of 0 if the named module is not present. On failure, `errno` is set.
I_FLUSH	Flushes all input and/or output queues, depending on the value of the `arg` parameter :
	FLSHR Flush read queues.
	FLUSHW Flush write queues.
	FLUSHRW Flush read and write queues.
I_GETSIG	Returns the events for which the calling process is currently registered to be sent a SIGPOLL signal. The events are returned as a bit mask pointed to by the `arg` parameter, where the events are those specified in the description of I_SETSIG. On failure, `errno` is set.
I_GRDOPT	Returns the current read mode setting in an int pointed to by the `arg` parameter. On failure, `errno` is set.
I_LINK	Connects two Streams, where the `filedes` parameter contains the file descriptor of the Stream connected to the multiplexing driver, and the `arg` parameter is the file descriptor of the Stream connected to another driver. The Stream designated by the `arg` parameter is connected below the multiplexing driver. I_LINK requires the multiplexing driver to send an acknowledgment message to the stream-head regarding the linking operation. It returns a multiplexer ID number (an identifier used to disconnect the multiplexer) on success, and a value of -1 on failure. On failure, `errno` is set.
I_LOOK	Retrieves the name of the module just below the stream-head of the Stream pointed to by the `filedes` parameter, and places it in a NULL-terminated character string pointed to by the `arg` parameter. On failure, `errno` is set.
I_NREAD	Counts the number of data bytes in data blocks in the first message on the stream-head read queue, and places this value in the location pointed to by the `arg` parameter. The return value for the command is the number of messages on the stream-head read queue. On failure, `errno` is set.
	For a pipe file, counts the number of data bytes left to read on the descriptor before the read function blocks.

I_NWRITE	Counts the number of data bytes that may be written on a pipe descriptor before the write function blocks. Similar to I_NREAD. For example, err=ioctl(pipeFD, I_NWRITE, &pending);.
I_PEEK	Allows a user to retrieve the information in the first message on the stream-head read queue without taking the message off the queue. The `arg` parameter points to a strpeek structure. I_PEEK returns a value of 1 if a message was retrieved, and returns a value of 0 if no message was found on the stream-head read queue.
I_POP	Removes the module just below the stream-head of the Stream pointed to by the `filedes` parameter. The `arg` parameter should be 0 in an I_POP request. On failure, `errno` is set.
I_PUSH	Pushes the module whose name is pointed to by the `arg` parameter onto the top of the current Stream, just below the stream-head. It then calls the open routine of the newly pushed module. On failure, `errno` is set.
I_RECVFD	Retrieves the file descriptor associated with the message sent by an I_SENDFD ioctl over a stream pipe. The `arg` parameter points to a data buffer large enough to hold the strrecvfd structure. If the message at the stream-head is a message sent by an I_SEND descriptor, a new user file descriptor is allocated for the file pointer contained in the message. The new file descriptor is placed in the `fd` field of the strrecvfd structure. The structure is copied into the user data buffer pointed to by the `arg` parameter. On failure, `errno` is set.
I_SENDFD	Requests the Stream associated with the file descriptor to send a message, containing a file pointer, to the stream-head at the other end of a stream pipe. The `arg` parameter must point to a file descriptor. I_SENDFD converts the `arg` parameter into the corresponding system file pointer. It allocates a message block and inserts the file pointer in the block. The user ID and group ID associated with the sending process are also inserted. This message is placed directly on the read queue for the stream-head at the other end of the stream pipe to which it is connected. On failure, `errno` is set.
I_SETBUF	Exchanges the buffer currently underlying the pipe for one of a different size. The pipe becomes empty and cleared with respect to data read or written. For example, err=ioctl(pipeFD, I_SETBUF, 8192);.
I_SETSIG	Informs the stream-head that the user wants the kernel to issue the SIGPOLL signal when a particular event has occurred on the Stream associated with the `filedes` parameter. I_SETSIG supports an asynchronous processing capability in STREAMS. The value of the `arg` parameter is a bit mask that specifies the events for which the user should be signaled. It is the bitwise OR of any combination of the following constants:
	S_INPUT Nonpriority message has arrived.
	S_HIPRI Priority message is present.
	S_OUTPUT The write queue is no longer full. There is room for sending (or writing) data downstream.
	S_MSG Stream signal message containing the SIGPOLL signal has reached the front of the stream-head read queue. On failure, `errno` is set.
I_SRDOPT	Sets the read mode using the `arg` parameter. Legal values for the `arg` parameter are:
	RNORM Byte-stream mode (the default)

	RMSGD	Message-discard mode
	RMSGN	Message-nondiscard mode

I_STR	Constructs an internal Stream ioctl message from the data pointed to by the `arg` parameter and sends that message downstream. I_STR blocks until the system responds with either a positive or negative acknowledgment message or until the request times out after some period of time. If the request fails, `errno` is set.
I_UNLINK	Disconnects the two Streams specified by the `filedes` and `arg` parameters. The `filedes` parameter is the file descriptor of the Stream connected to the multiplexing driver. The descriptor must correspond to the Stream on which the ioctl I_LINK command was issued to link the Stream below the multiplexing driver. The `arg` parameter is the multiplexer ID number that was returned by the I_LINK call. If the `arg` parameter is -1, all Streams which were linked to the `filedes` parameter are disconnected. As in I_LINK, it requires the multiplexing driver to acknowledge the unlink. On failure, `errno` is set.
SIOCATMARK	Supported for TCP sockets. Returns a nonzero value if the socket's read pointer is currently at the out-of-band mark. A zero value indicates the read pointer is not at the out-of-band mark. The value is returned in the third parameter of ioctl.
SIOCDGRAMSIZE	Supported for RAWIP sockets (NetWare specific). Returns the size of the datagram to be sent, in bytes. The value is returned in the third parameter of ioctl.

See Also

pipe (page 232), poll (unsupported), putmsg (NDK: NetWare Server Protocol Libraries for C)(unsupported), open (page 229), read (page 234), socket (NDK: NetWare Server Protocol Libraries for C)(unsupported), write (page 246)

isatty

Tests whether the specified handle refers to a screen

Local Servers: nonblocking

Remote Servers: nonblocking

Classification: POSIX

Platform: NLM

Service: Operating System I/O

Syntax

```
#include <unistd.h>

int isatty  (
   int   handle);
```

Parameters

handle

   (IN) Specifies a file handle.

Return Values

isatty n returns a value of 0 if the device or file is not a character device; otherwise, a nonzero is returned.

If an error occurs, `errno` can be set to:

4	EBADF	Bad file number.

Remarks

The isatty function tests if the opened file or device referenced by the file handle is a character device (namely, the console).

See Also

lock

Locks a portion of a file

Local Servers: blocking

Remote Servers: blocking

Platform: NLM

Service: Operating System I/O

Syntax

```
#include <nwfileio.h>

int lock  (
   int    fildes,
   LONG   offset,
   LONG   length);
```

Parameters

handle

(IN) Specifies a file handle.

offset

(IN) Specifies the starting byte that is to be locked.

length

(IN) Specifies the amount of data (in bytes) to be locked.

Return Values

Returns a value of 0 if successful, and a value of -1 when an error occurs.

If an error occurs, `errno` is set to:

4	EBADF	Bad file number.

Remarks

lock locks the amount of data specified by the `length` parameter in the file specified by the `handle` parameter, starting at the byte specified by the `offset` parameter in the file.

lock prevents other open handles from reading or writing into the locked region until an unlock has been done for this locked region of the file. All locked regions of a file must be unlocked before a file is closed.

See Also

open (page 229), sopen (page 238), unlock (page 244)

lseek

Sets the current file position

Local Servers: nonblocking

Remote Servers: nonblocking

Classification: POSIX

Platform: NLM

Service: Operating System I/O

Syntax

```
#include <unistd.h>

LONG lseek  (
   int    handle,
   LONG   offset,
   int    origin);
```

Parameters

handle

   (IN) Specifies a file handle.

offset

   (IN) Specifies the relative offset from a file position.

origin

   (IN) Specifies the seek starting point.

Return Values

On success, lseek returns the new current file position in a system-dependent manner. A value of 0 indicates the start of a file. If an error occurs, lseek returns 0xFFFFFFFF, and `errno` is set to:.

4	EBADF	Bad file number.

If lseek does not complete successfully, `NetWareErrno` is also set.

Remarks

The file is referenced using the specified file handle.

The value of the `offset` parameter is used as a relative offset from a file position determined by the value of the `origin` parameter. An absolute offset can be from 0 to 2^{32}. It must be unsigned long, which cannot be negative.

The new file position is determined in a manner dependent upon the value of the `origin` parameter, which can have one of three possible values (defined in the STDIO.H header file):

SEEK_SET	The new file position is computed relative to the start of the file.
SEEK_CUR	The new file position is computed relative to the current file position.
SEEK_END	The new file position is computed relative to the end of the file.

The files's position can be set to a position outside of the bounds of the file.

See Also

close (page 205), creat (page 207), dup (page 209), dup2 (page 211), eof (page 213), filelength (page 216), fileno (page 271), fstat (page 217), isatty (page 223), open (page 229), read (page 234), sopen (page 238), tell (page 242), write (page 246)

Example

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>

main()
{
   int   fh;
   if((fh = open ("test.dat",O_RDWR | O_CREAT | O_TRUNC,0)) < 0)
   {
      printf ("could not open file\r\n");
      exit(0);
   }
   write (fh,"1234567890",10);
   if(lseek (fh,1,SEEK_SET) < 0)
   {
      printf ("error on seek 1\r\n");
      goto end;
   }
   write (fh,"a",1);
   if(lseek(fh,-2,SEEK_END) < 0)
   {
      printf ("error on seek 2\r\n");
      goto end;
   }
   write (fh,"b",1);
   if(lseek (fh,-5,SEEK_CUR) < 0)
   {
      printf ("error on seek 3\r\n");
      goto end;
   }
   write (fh,"c",1);
```

```
            end:
            close (fh);
            getch ();
    }
```

open

Opens a file or stream

Local Servers: blocking

Remote Servers: blocking

Classification: POSIX

Platform: NLM

Service: Operating System I/O

Syntax

```
#include <fcntl.h>

int open  (
   const char   *path,
   int           oflag,
   ...);
```

Parameters

path

(IN) Points to the name of the file to open (considered within the context of the currently set namespace).

oflag

(IN) Specifies the access mode.

Return Values

When an error occurs while opening the file, a value of -1 is returned. Otherwise, an integer (not equal to -1), known as the file handle, is returned to be used with the other functions that operate on the file.

If an error occurs, `errno` can be set to:

Decimal	Constant	Description
1	ENONENT	No such file.
6	EACCES	Permission denied.
9	EINVAL	Invalid argument.

When an error occurs, `NetWareErrno` is set to:

Decimal	Hex	Constant
108	(0x6C)	ERR_BAD_ACCESS

Decimal	Hex	Constant
152	(0x98)	ERR_INVALID_VOLUME
156	(0x9C)	ERR_INVALID_PATH

Remarks

This function also works on the DOS partition.

This function allows for as many open files as there is available memory. The `path` parameter supplies the name of the file to be opened. The file is accessed according to the access mode specified by the `oflag` parameter.

The access mode is established as a combination of the bits defined in the FCNTL.H header file. The following bits can be set:

O_RDONLY	Permits the file to be only read.
O_WRONLY	Permits the file to be only written.
O_RDWR	Permits the file to be both read and written.
O_APPEND	Causes each record that is written to be written at the end of the file.
O_CREAT	Has no effect when the file indicated by the file name parameter already exists; otherwise, the file is created.
O_TRUNC	Causes the file to be truncated to contain no data when the file exists; has no effect when the file does not exist. O_TRUNC must be ORed with write access to truncate a file: O_TRUNC | O_RDWR O_TRUNC | O_WRONLY
O_BINARY	Causes the data to be transmitted unchanged. (Text mode for first-level handles is not supported. In text mode, carriage- return/line-feed pairs are translated to line feeds on input, and line feeds are translated to carriage-return/line-feed pairs on output.)

O_CREAT must be specified when the file does not exist and it is to be written.

An optional third parameter, int `permission`, is used when the file is to be created (O_CREAT is specified) to set file permissions. File permissions are set according to the value contained in the `permission` parameter. The access permissions for the file is specified as a combination of bits (defined in the SYS\STAT.H header file).

S_IWRITE	The file is writable.
S_IREAD	The file is readable.

The `permission` parameter can be specified as S_IWRITE, S_IREAD or S_IWRITE|S_IREAD. Specifying 0 also makes a file both writable and readable.

When opening a UNIX Stream file, access must be constructed from O_NDELAY and either O_RDONLY, O_WRONLY, or O_RDWR. Other flag values are not applicable to Stream devices

and have no effect on them. The value of O_NDELAY affects the operation of Stream drivers and certain function calls (read, getmsg, putmsg, and write). For drivers, the implementation of O_NDELAY is device-specific. Each Stream device driver can treat this option differently.

SetCurrentNameSpace sets the name space which is used for parsing the path input to this function.

See Also

close (page 205), creat (page 207), dup (page 209), dup2 (page 211), eof (page 213), filelength (page 216), fileno (page 271), fstat (page 217), isatty (page 223), lseek (page 226), read (page 234), sopen (page 238), tell (page 242), write (page 246)

Example

```
#include <stddef.h>
#include <fcntl.h>
#include <errno.h>

main()
{
   int fh,size;
   char buffer[] = {"a text record to be written\n" };
   fh = open ("test.dat",O_WRONLY | O_CREAT | O_TRUNC,0);
   printf ("handle: %d\n\r",fh);
   if(fh == EFAILURE)
   {
      printf ("could not open file\r\n");
      goto end;
   }
   printf ("%d\n\r",tell(fh));
   size = write (fh,buffer,sizeof(buffer));
   if(size < 29)
   {
      printf ("could not write to file\r\n");
      goto end;
   }
   printf ("%d\r\n",tell(fh));
   close (fh);
   end:
   printf ("%d\r\n",errno);
   getch ();
}
```

pipe

Creates an inter-process channel.

Local Servers: blocking

Remote Servers: N/A

Classification: UNIX (nonstandard)

NetWare Server: 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM

Service: Operating System I/O

Syntax

```
#include <unistd.h>

int pipe (
   int   fildes[2]);
```

Parameters

fildes

   (OUT) Specifies the file descriptions for the ends of the newly created pipe.

Return Values

pipe returns zero upon successful completion. Otherwise, -1 is returned and `errno` indicates the occurring error.

If an error occurs, `errno` can be set to:

EMFILE	More than (OPEN_MAX)-2 file descriptors are already in use.
ENFILE	The number of simultaneously open files in the system would exceed a system-imposed limit.

Remarks

pipe creates a pipe and places two file descriptions (into `fildes`[0] and `fildes`[1]) for the read and write ends of the pipe. Their integer values shall be the two lowest that available at the time pipe is called. O_NONBLOCK and FD_CLOEXEC, which can be set by calling fcntl, should be clear on both file descriptors.

Data can be written to `fildes`[1] and read from `fildes`[0]. A read on `fildes`[0] accesses the data written to `fildes`[1] on a first-in-first-out basis.

A process has the pipe open for reading if it has a file descriptor open that refers to the read end, `fildes`[0]. A process has the pipe open for writing if it has a file descriptor open that refers to the write end, `fildes`[1].

Upon successful completion, pipe updates the `st_atime`, `st_ctime`, and `st_mtime` fields of the new pipe.

See Also

fcntl (page 214), ioctl (page 219), poll (unsupported), putmsg (unsupported) (NDK: NetWare Server Protocol Libraries for C), open (page 229), read (page 234), socket (unsupported) (NDK: NetWare Server Protocol Libraries for C), write (page 246)

read

Reads data from a file or stream

Local Servers: blocking

Remote Servers: blocking

Classification: POSIX

Platform: NLM

Service: Operating System I/O

Syntax

```
#include <unistd.h>

LONG read  (
   int    handle,
   char  *buffer,
   LONG   len);
```

Parameters

handle

   (IN) Specifies a file handle.

buffer

   (OUT) Points to a buffer to receive the data.

len

   (IN) Specifies the number of bytes to read.

Return Values

read returns the number of bytes of data transmitted from the file to the buffer. Normally, this is the number given by the `len` parameter. When the end-of- file is encountered before the read completes, the return value is less than the number of bytes requested.

A value of -1 is returned when an input/output error is detected. If an error occurs, `errno` can be set to:

4	EBADF	Bad file number.

If read does not complete successfully, `NetWareErrno` is set.

Remarks

This function also works on the DOS partition.

The read function returns the number of bytes of data transmitted from the file to the buffer.

The `handle` value is returned by the open, sopen, creat, or fileno function. The access mode must have included either O_RDONLY or O_RDWR when the open or sopen function was invoked. The data is read starting at the current file position for the file in question. This file position can be determined with the tell function and can be set with the lseek function.

A read from a Stream file can operate in three different modes: byte-stream mode, message-nondiscard mode, and message-discard mode. The default is byte-stream mode. This can be changed using the I_SRDOPT ioctl request, and can be tested with the I_GRDOPT ioctl. In byte-stream mode, the read function retrieves data from the Stream until it has received `nbyte` bytes, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

In Stream message-nondiscard mode, the read function retrieves data until it has read `nbytes` bytes or until it reaches a message boundary. If the read function does not retrieve all the data in the message, the remaining data are placed on the Stream, and can be retrieved by the next read or getmsg call. Message-discard mode also retrieves data until it has retrieved `nbyte` bytes or it reaches a message boundary. However, unread data remaining in a message after the read function returns are discarded and are not available for a subsequent read or getmsg.

When reading from a Stream file, handling of zero-byte messages is determined by the current read mode setting. In byte-stream mode, the read function accepts data until it has read `nbyte` bytes, or until there is no more data to read or until a zero-byte message block is encountered. The read function returns the number of bytes read and places the zero-byte message back on the Stream to be retrieved by the next read or getmsg. In the two other modes, a zero-message returns a value of 0 and the message is removed from the Stream. When a zero-byte message is read as the first message on a Stream, a value of 0 is returned regardless of the read mode.

A read from a Stream file can only process data messages. It cannot process any type of protocol message and fails if a protocol message is encountered at the stream head.

A read from a Stream file also fails if an error message is received at the stream head. In this case, `errno` is set to the value returned in the error message. If a hang up occurs on the Stream being read, the read function continues to operate normally until the stream head read queue is empty. Thereafter, it returns 0.

See Also

close (page 205), creat (page 207), dup (page 209), dup2 (page 211), eof (page 213), filelength (page 216), fileno (page 271), fstat (page 217), isatty (page 223), lseek (page 226), open (page 229), sopen (page 238), tell (page 242), write (page 246)

Example

```
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <stddef.h>
#include <stdio.h>
#include <string.h>
#include <dirent.h>
#include <nwconio.h>

#define BUFFER_SIZE 512
```

```
main()
{
   int     handle, rc, i, ch = 0;
   char    buffer[BUFFER_SIZE];
   handle = open ("test.dat", O_RDONLY, 0 );
   if (handle < 0)
   {
      printf ("\r%s\r\n\n",strerror(errno));
      exit (0);
   }
   while (1)
   {
      if ((rc = read (handle, buffer, BUFFER_SIZE)) <= 0)
      {
         lseek (handle,0,SEEK_SET);
         rc = 0;
      }
      for (i = 0; i < rc; i++)
      {
         putchar (buffer[i] );
         if (kbhit ())
         {
         if ((ch = getch()) == 0x03)
         {
            printf ("^C" );
            close (handle );
            exit (0);
         }
         else if(ch == 'l')
         {
            printf ("\r\n\n***** lock %s\r\n\n",
            lock (handle,1,10) ? "failed" : "succeeded");
            getch ();
         }
         else if(ch == 'u')
         {
            printf ("\r\n\n***** unlock %s\r\n\n",
            unlock(handle,1,10) ? "failed" : "succeeded");
            getch ();
         }
         else getch (); /* Pause*/
         }
      }
   }
}
```

setmode

Sets, at the operating system level, the translation mode to the specified value

Local Servers: nonblocking

Remote Servers: nonblocking

Platform: NLM

Service: Operating System I/O

Syntax

```
#include <nwfileio.h>

int setmode  (
   int   handle,
   int   mode);
```

Parameters

handle

   (IN) Specifies a file handle.

mode

   (IN) Specifies the translation mode.

Return Values

If successful, the setmode function returns the previous mode that was set for the file. Otherwise, a value of -1 is returned. When an error has occurred, the global variable `errno` contains a value indicating the type of error that has been detected.

Remarks

The setmode function sets the translation mode to be the value of mode for the file whose file handle is given by handle. The `mode` parameter can contain the following value:

• O_BINARY—Data is read or written unchanged.

See Also

close (page 205), creat (page 207), eof (page 213), filelength (page 216), fileno (page 271), fstat (page 217), isatty (page 223), lseek (page 226), open (page 229), read (page 234), sopen (page 238), tell (page 242), write (page 246)

sopen

Opens a file or stream for shared access

Local Servers: blocking

Remote Servers: blocking

Platform: NLM

Service: Operating System I/O

Syntax

```
#include <nwfileio.h>

int sopen  (
   const char   *path,
   int           oflag,
   int           shflag,
   ...);
```

Parameters

path

   (IN) Points to the path of the file to open.

oflag

   (IN) Specifies the access mode.

shflag

   (IN) Specifies the sharing mode of the file.

Return Values

When an error occurs while opening the file, a value of –1 is returned. Otherwise, an integer (not equal to –1), known as the file handle, is returned to be used with the other functions that operate on the file.

When an error occurs, `errno` is set to:

Decimal	Constant	Description
1	ENOENT	No such file.
6	EACCESS	Permission denied.
9	EINVAL	Invalid argument.

When an error occurs, `NetWareErrno` is set to:

Decimal	Hex	Constant
107	(0x6B)	ERR_BAD_SHFLAG
108	(0x6C)	ERR_BAD_ACCESS
152	(0x98)	ERR_INVALID_VOLUME
156	(0x9C)	ERR_INVALID_PATH

NOTE: A problem was introduced in the 4.01d version of clib.nlm. It manifests itself when sopen is called multiple times by the same thread.

In clib.nlm versions prior to 4.01d, a valid file handle will be returned from sopen when a file is opened multiple times by the same thread. In the 4.01d version of clib.nlm (and later versions) an error occurs and returns -1 while errno is set to EINUSE.

Beginning with version 4.10d (notice this is 4.1 and not 4.01) PTR 2/1/95, you can pass the NWSH_PRE_401D_COMPAT bit (0x80000000) in as part of the share flags passed to sopen to get functionality of clib versions prior to 4.01d (if you have experienced problems with multiple threads opening the same file multiple times).

As an alternative to the above work around, you can pass in a CLIB_OPT switch on the command line of the .nlm you are using. CLIB_OPT/U86414 will provide the same functionality as passing in the share flags with the additional bits ORed in. This is only available on clib.nlm version 4.10d PTF.

Remarks

This function also works on the DOS partition.

The path of the file to be opened is given by the `path` parameter. The file is accessed according to the access mode specified by the `oflag` parameter.

The access mode is established as a combination of the bits defined in the FCNTL.H header file. The following bits can be set:

O_RDONLY	Permits the file to be only read.
O_WRONLY	Permits the file to be only written.
O_RDWR	Permits the file to be both read and written.
O_APPEND	Causes each record that is written to be written at the end of the file.
O_CREAT	Has no effect when the file indicated by the filename parameter already exists; otherwise, the file is created.
O_TRUNC	Causes the file to be truncated to contain no data when the file exists; has no effect when the file does not exist. O_TRUNC must be ORed with write access to truncate a file:

> O_TRUNC | O_RDWR
>
> O_TRUNC | O_WRONY

O_BINARY	Causes the data to be transmitted unchanged. (Text mode for first-level handles is not supported. In text mode, carriage- return/line-feed pairs are translated to line feeds on input, and line feeds are translated to carriage-return/line-feed pairs on output.)

O_CREAT must be specified when the file does not exist and it is to be written.

The shared access for the file is established by the combination of bits set in the `shflag` parameter where the following values are defined in NWFATTR.H.

SH_COMPAT	Sets compatibility mode.
SH_DENYRW	Prevents read or write access to the file.
SH_DENYWR	Prevents write access of the file.
SH_DENYRD	Prevents read access to the file.
SH_DENYNO	Permits both read and write access to the file.

NOTE: If a new file is created by this function, the share flag is ignored.

An optional fourth parameter, int `permission`, is used when the file is to be created (O_CREAT is specified) to set file permissions. File permissions are set according to the value contained in the `permission` parameter. The access permissions for the file is specified as a combination of bits (defined in the SYS\STAT.H header file).

S_IWRITE	The file is writable.
S_IREAD	The file is readable.

The `permission` parameter can be specified as S_IWRITE, S_IREAD or S_IWRITE|S_IREAD. Specifying 0 also makes a file both writable and readable.

See Also

Example

```
#include <errno.h>
#include <fcntl.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <nwbindry.h>
#include <nwfattr.h>
#include <nwerrno.h>
#include <nwconn.h>
#include <nwfileio.h>
#include <unistd.h>
```

```c
#define BUFFER_SIZE 512
main()
{
   int ccode, handle;
   errno=0;
   printf("Login %s\r\n\n", LoginToFileServer("ADMIN",//
      supervisor for NW 3.x OT_USER,"\x0")?"Failed":"Succeeded");
   handle=sopen("test.c", O_RDWR, SH_DENYNO, 0);
   if (handle==-1)
   {
      printf("\r%s\r\n\n",strerror(errno));
      close(handle);
      Logout();
      exit(0);
   }
   ccode=lock(handle, 1, 10);
   if (ccode)
   {
      printf("\n***lock failed\n");
      printf("r%s\r\n\n", strerror(errno));
   }
   else
      printf("\n***lock succeeded\n");
   getch();
   ccode=unlock(handle, 1, 10);
   if (ccode)
   {
      printf("\n***unlock failed\n");
      printf("\r%s\r\n\n", strerror(errno));
   }
   else
      printf("\n***unlock succeeded\n");
   close(handle);
   Logout();
}
```

tell

Determines the current file position

Local Servers: nonblocking

Remote Servers: nonblocking

Platform: NLM

Service: Operating System I/O

Syntax

```
#include <nwfileio.h>

LONG tell  (
   int   fildes);
```

Parameters

handle

   (IN) Specifies a file handle.

Return Values

When an error occurs, a value of -1 is returned. Otherwise, the current file position is returned in a system-dependent manner. A value of 0 indicates the start of the file.

If an error occurs, `errno` is set to:

4	EBADF	Bad file number.

Remarks

The `handle` value is the file handle returned by a successful execution of the open, sopen, creat, or fileno function. This function can be used in conjunction with lseek to reset the current file position.

See Also

Example

```
#include <fcntl.h>

LONG   position;
```

```
int     handle;
position = tell (handle);
```

unlock

Unlocks a region of previously locked data in a file

Local Servers: blocking

Remote Servers: blocking

Platform: NLM

Service: Operating System I/O

Syntax

```
#include <nwfileio.h>

int unlock  (
   int    fildes,
   LONG   offset,
   LONG   length);
```

Parameters

handle

   (IN) Specifies a file handle.

offset

   (IN) Specifies the starting byte.

length

   (IN) Specifies the amount of data (in bytes).

Return Values

unlock returns a value of 0 if successful and a value of -1 when an error occurs.

If an error occurs, `errno` can be set to:

Decimal	Constant	Description
4	EBADF	Bad file number.
19	EWRNGKND	The region was not locked.

If unlock does not complete successfully, `NetWareErrno` is set.

Remarks

The unlock function unlocks the region of the file previously locked with the lock function that specified the same offset as the call to unlock. If the file does not have a locked region starting at `offset`, the unlock function returns a value of -1 and sets `errno` to EWRNGKND. All locked regions of a file should be unlocked before a file is closed.

See Also

dup (page 209), dup2 (page 211), lock (page 224), open (page 229), sopen (page 238)

write

Writes data (blocks even if writing to the screen)

Local Servers: blocking

Remote Servers: blocking

Classification: POSIX

Platform: NLM

Service: Operating System I/O

Syntax

```
#include <unistd.h>

LONG write  (
   int     handle,
   char   *buffer,
   LONG    len);
```

Parameters

handle

   (IN) Specifies a file handle.

buffer

   (IN) Point to the address at which to start transmitting data.

len

   (IN) Specifies the number of bytes transmitted.

Return Values

write returns the number of bytes of data transmitted to the file. When there is no error, this is the number given by the `len` parameter. In the case of an error, such as there being no space available to contain the file data, the return value is less than the number of bytes transmitted. A value of -1 is returned in the case of output errors.

If an error occurs, `errno` can be set to:

4	EBADF	Bad file number.

If write does not complete successfully, `NetWareErrno` is set.

Remarks

This function also works on the DOS partition.

The `handle` value is returned by open, sopen, or creat. The access mode must have included either O_WRONLY or O_RDWR when open or sopen was invoked. The data is written to the file at the end when the file was opened with O_APPEND included as part of the access mode; otherwise, it is written at the current file position for the file in question. This file position can be determined with tell and can be set with lseek.

For Stream files, the operation of write is determined by the values of the minimum and maximum n-byte range (packet size) accepted by the Stream. These values are contained in the topmost Stream module. Unless the user pushes the topmost module, these values cannot be set or tested from user level.

- If n-byte falls within the packet size range, n-byte bytes are written.
- If n-byte does not fall within the range and the minimum packet size value is zero, write breaks the buffer into maximum packet size segments prior to sending the data downstream (the last segment can contain less than the maximum packet size).
- If nbyte does not fall within the range and the minimum value is nonzero, write fails with `errno` set to ERANGE. Writing a zero-length buffer (n-byte is 0) sends zero bytes with zero bytes returned.

For Stream files, if O_NDELAY is not set and the Stream cannot accept data (the stream write queue is full due to internal flow control conditions), write blocks until data can be accepted. O_NDELAY prevents a process from blocking due to flow control conditions. If O_NDELAY is set and the Stream cannot accept data, write fails. If O_NDELAY is set and part of the buffer has been written when a condition in which the Stream cannot accept additional data occurs, write terminates and returns the number of bytes written.

See Also

close (page 205), creat (page 207), dup (page 209), dup2 (page 211), eof (page 213), filelength (page 216), fileno (page 271), fstat (page 217), isatty (page 223), lseek (page 226), open (page 229), read (page 234), sopen (page 238), tell (page 242)

Example

```
#include    <stddef.h>
#include    <unistd.h>
#include    <fcntl.h>
#include    <errno.h>
#include    <string.h>

main()
{
   int fh,size;
   char buffer[] = { "NLM test file" };
   size = strlen(buffer);
   errno = 0;
   if((fh = creat ("test.dt1",0)) != -1)
   {
      printf ("open 1:  %s\r\n",strerror(errno));
   }
   else
   {
```

```
      if(write (fh,buffer,size) < size)
      printf ("write 1:  %s\r\n",strerror(errno));
      printf ("file length 1: %d\r\n",filelength(fh));
      if(close (fh) < 0)
      printf ("close 1:  %s\r\n",strerror(errno));
   }
   if((fh = creat ("test.dt2",0)) != -1)
   {
      printf ("open 2:  %s\r\n",strerror(errno));
   }
   else
   {
      if(write (fh,buffer,size) < size)
      printf ("write 2:  %s\r\n",strerror(errno));
      printf ("file length 2: %d\r\n",filelength(fh));
      if(close (fh) < 0)
      printf ("close 2:  %s\r\n",strerror(errno));
   }
   if((fh = creat ("test.dt3",0)) != -1)
   {
      printf ("open 3:  %s\r\n",strerror(errno));
   }
   else
   {
      if(write (fh,buffer,size) < size)
      printf ("write 3:  %s\r\n",strerror(errno));
      printf ("file length 3: %d\r\n",filelength(fh));
      if(close (fh) < 0)
      printf ("close 3:  %s\r\n",strerror(errno));
   }
   printf ("\r\n exit:  %s\r\n",strerror(errno));
   getch ();
}
```

Stream I/O Concepts

<div style="text-align: right; font-size: 3em;">18</div>

This documentation describes Stream I/O, its functions, and features.

NOTE: The streams discussed here are standard files, not to be confused with UNIX STREAMS or STREAMS.

Developed by USL, the UNIX based STREAMS facility, or mechanism, is a collection of system calls, kernel resources, and kernel utility routines. The STREAMS mechanism creates, uses, and dismantles a Stream, which is a full-duplex processing and data transfer path between a driver in kernel space and a process in user space; a STREAM consists of three basic components: a stream head, stream modules (protocol stacks), and a stream driver.

Stream I/O functions can be used for "standard" read and write file operations. Data can be transmitted as characters, strings, or blocks of memory.

A stream is the name given to a second-level file that has been opened for data transmission. When a stream is opened, a pointer to a FILE structure is returned. This pointer is used to reference the stream when other functions are subsequently invoked.

18.1 Stream I/O Functions

These are the functions for handling stream input and output:

clearerr	Clears end-of-file error indicators for a stream.
fclose	Closes a stream file.
fcloseall	Closes all open stream files except stdin, stdout, and stderr.
fdopen	Associates a stream with a file handle that represents an open file or device.
feof	Tests the end-of-file indicator for a stream.
ferror	Tests the error indicator for a stream.
fflush	Flushes the output buffer of a stream.
fgetc	Returns the next character from the input stream.
fgetchar	Returns the next character from the input stream pointed to by stdin.
fgetpos	Retrieves the current position of a stream.
fgets	Gets a string of characters from a stream and stores them in an array.
fileno	Returns the file handle for a stream.
flushall	Clears all buffers associated with input streams and writes any buffers associated with output streams.
fopen	Opens a file and associates a stream with it.
fprintf	Writes output to a stream under format control.
fputc	Writes a character to the output stream.

fputs	Writes a string to an output stream.
fread	Reads data from a stream.
freopen	Opens a file and associates a previously opened stream with it.
fscanf	Scans input from a stream under format control.
fseek	Changes the read/write position of a stream.
fsetpos	Sets the current position of a stream.
ftell	Returns the current read/write position of a stream.
fwrite	Writes elements to a stream.
getc	Gets the next character from a stream.
getchar	Gets the next character from stdin.
gets	Gets a string from a stream and stores it in an array.
printf	Writes output to the stream designated by `stdout`.
putc	Writes a character to an output stream.
putchar	Writes a character to an output stream.
puts	Writes a specified character string to an output stream and appends a newline character to the output.
rewind	Sets the stream position indicator to the beginning of the file.
scanf	Scans input from a stream.
setbuf	Associates a buffer with a stream after the stream is open and before it has been read or written to.
setvbuf	Associates a buffer with a stream after the stream is open and before it has been read or written to.
tmpfile	Creates a temporary binary file.
ungetc	Pushes a character back onto the specified input stream.
vfprintf	Writes output to a stream under format control.
vfscanf	Scans input from a stream under format control.
vprintf	Writes output to a stream under format control.
vscanf	Scans input from the stream designated by `stdin`.

Stream I/O Functions

19

This documentation alphabetically lists the Stream I/O functions and describes their purpose, syntax, parameters, and return values.

clearerr

Clears the end-of-file and error indicators for a stream (function or macro)

Local Servers: blocking

Remote Servers: blocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

void clearerr  (
   FILE   *fp);
```

Parameters

fp

   (IN) Points to the file to be cleared.

Remarks

The clearerr function or macro clears the end-of-file and error indicators for the file pointed to by fp. These indicators are cleared only when the file is opened or by an explicit call to the clearerr or rewind functions.

See Also

feof (page 259), ferror (page 261), perror (*NDK: NLM Development Concepts, Tools, and Functions*)

clearerr Example

```
#include <stdio.h>

main ()
{
   FILE   *fp;
   int    c;
   fp=fopen("testfile", "wt");
   if (ferror (fp) )
   {                                 /* If error,*/
      clearerr (fp);                 /* clear the error */
      fputc (c, fp);                 /* and retry it */
   }
}
```

fclose

Closes a stream file

Local Servers: blocking

Remote Servers: blocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

int fclose  (
    FILE   *fp);
```

Parameters

fp

   (IN) Points to the file to close.

Return Values

The fclose function returns a value of 0 if the file was successfully closed or nonzero if any errors were detected. When an error has occurred, `errno` is set.

Remarks

This function also works on the DOS partition.

The fclose function closes the file pointed to by `fp`. If there is unwritten buffered data for the file, it is written before the file is closed. Unread buffered data is discarded. If the associated buffer was automatically allocated, it is deallocated.

See Also

fcloseall (page 255), fdopen (page 256), fopen (page 274), freopen (page 285)

fcloseall

Closes all open stream files except stdin, stdout and stderr

Local Servers: blocking

Remote Servers: blocking

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

int fcloseall  (void);
```

Return Values

The fcloseall function returns the number of files that were closed, if no errors were encountered. When an error occurs, EOF is returned.

Remarks

The fcloseall function closes all open stream files except stdin, stdout, and stderr. Files closed includes files created (and not yet closed) by fdopen, fopen, and freopen.

See Also

fcloseall Example

```
#include <stdio.h>

main ()
{
   printf ("The number of files closed is %d\n", fcloseall () );
}
```

fdopen

Associates a stream with a file handle that represents an open file or device

Local Servers: nonblocking

Remote Servers: nonblocking

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

FILE *fdopen  (
   int           handle,
   const char    *mode);
```

Parameters

handle

> (IN) Specifies a file handle.

mode

> (IN) Points to a file mode.

Return Values

The fdopen function returns a pointer to the object controlling the stream. This pointer must be passed as a parameter to subsequent functions for performing operations on the file. If the open operation fails, fdopen returns a NULL pointer. When an error has occurred, `errno` is set.

Remarks

The fdopen function associates a stream with `handle`, which represents an opened file or device. The handle was returned by a creat or open function. `mode` must match the mode with which the file or device was originally opened.

If `mode` does not match the access flags used in opening the file originally, errno will be set to EINVAL and fdopen will fail. This includes the mode accesses read, write, append, and binary. See argument `oflag` for function open.

The fdopen function opens the file whose name is the string pointed to by `filename`, and associates a stream with it. The argument `mode` points to a string beginning with one of the following sequences:

r	Opens file for reading.
w	Creates file for writing, or truncates to zero length; uses default file translation.

a	Appends; opens or creates text file for writing at end-of-file; uses default file translation.
rb	Opens binary file for reading.
rt	Opens text file for reading.
wb	Creates binary file for writing, or truncates to zero length.
wt	Creates text file for writing, or truncates to zero length.
ab	Appends; opens or creates binary file for writing at end-of-file.
at	Appends; opens or creates text file for writing at end-of-file.
r+	Opens file for update (reading and/or writing); uses default file translation.
w+	Creates file for update, or truncates to zero length; uses default file translation.
a+	Appends; opens or creates file for update, writing at end-of-file; uses default file translation.
r+b	Opens binary file for update (reading and/or writing).
r+t	Opens text file for update (reading and/or writing).
w+b	Creates binary file for update, or truncates to zero length.
w+t	Creates text file for update, or truncates to zero length.
a+b	Appends; opens or creates binary file for update, writing at end-of-file.
a+t	Appends; opens or creates text file for update, writing at end-of-file.
rb+	Opens binary file for update (reading and/or writing).
rt+	Opens text file for update (reading and/or writing).
wb+	Creates binary file for update, or truncates to zero length.
wt+	Creates text file for update, or truncates to zero length.
ab+	Appends; opens or creates binary file for update, writing at end-of-file.
at+	Appends; opens or creates text file for update, writing at end-of-file.

See Also

fdopen Example

This example shows how to reverse the effects of redirecting stdin.

```
#include <stdio.h>
int func (char *filepath )
{
   int     fd, stdin_fd;
   char    line[512];
   FILE    *fp;
```

```
    stdin_fd = fileno(stdin);   /*save descriptor for 'stdin' */
    fd   = dup(stdin_fd);

    if (fd == -1)
        return -1;                /* failed to duplicate input descriptor
*/

    /* use the duplicated descriptor to redirect input... */
    fp = fdopen (fd, "r");

    if (!fp)
        return -2;                /* failed to open duplicated descriptor
*/

    stdin = freopen (filepath, "r", fp);

    if (!stdin)
        return -3;                /* failed to redirect stream input */

    /* use redirected stream (example)... */
    while (gets(line))
        printf("%s\n", line);

    /* UNDO: now undo the effects of redirecting input... */
    fclose(stdin);

    stdin = fdopen(stdin_fd, "r");

    if (!stdin)
        return -4;                /* failed to reestablish 'stdin' */

    return 0;
}
```

If `stdin` is redirected by a console command such as

```
LOAD NLM-NAME /(CLIB_OPT)/<filename>
```

you can likewise return the standard input to the keyboard by using the statements following the "UNDO" comment in the example.

feof

Tests the end-of-file indicator for a stream (function or macro)

Local Servers: nonblocking

Remote Servers: nonblocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

int feof  (
   FILE   *fp);
```

Parameters

fp

(IN) Points to the file to be tested.

Return Values

feof returns nonzero if the EOF indicator is set for `fp`.

Remarks

The feof function or macro tests the end-of-file indicator for the file pointed to by `fp`. Because this indicator is set when an input operation attempts to read past the end of the file, feof detects the end of the file only after an attempt is made to read beyond the end of the file. Thus, if a file contains 10 lines, feof does not detect the end of the file after the tenth line is read; it detects the end of the file once the program attempts to read more data.

See Also

feof Example

```
#include <stdio.h>

main ()
{
   FILE *fp;
   char buffer[100];
   fgets (buffer, sizeof (buffer), fp);
```

```
        while (! feof (fp) )
        {
           process_record (buffer);
           fgets (buffer, sizeof (buffer), fp);
        }
     }
```

ferror

Tests the error indicator for a stream (function or macro)

Local Servers: nonblocking

Remote Servers: nonblocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

int ferror  (
   FILE   *fp);
```

Parameters

fp

   (IN) Points to the file to be tested.

Return Values

ferror returns nonzero if the error indicator is set for `fp`.

Remarks

The ferror function or macro tests the error indicator for the file pointed to by `fp`.

See Also

clearerr (page 253), feof (page 259), strerror (*NDK: NLM Development Concepts, Tools, and Functions*)

ferror Example

```
#include <stdio.h>

main ()
{
   FILE    *fp;
   int     c;
   c = fgetc (fp);
   if (ferror (fp) )
   {                      /* if end-of-file */
      fclose  (fp);       /* close the file */
```

```
            c = EOF;
        }
    }
```

fflush

Flushes the output buffer of a stream

Local Servers: blocking

Remote Servers: blocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

int fflush  (
   FILE   *fp);
```

Parameters

fp

   (IN) Points to the file to be flushed.

Return Values

The fflush function returns nonzero if a write error occurs, and returns zero otherwise. If an error occurs, `errno` is set.

Remarks

If the file pointed to by `fp` is open for output or update, the fflush function causes any unwritten data to be written to the file. If the file pointed to by `fp` is open for input or update, the fflush function undoes the effect of any preceding ungetc operation on the stream. If the value of `fp` is NULL, all open files are flushed.

See Also

fgetc

Returns the next character from the input stream

Local Servers: blocking

Remote Servers: blocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

int fgetc  (
   FILE   *fp);
```

Parameters

fp

   (IN) Points to the file.

Return Values

The fgetc function returns the next character from the input stream pointed to by `fp`. If the stream is at end-of-file, the EOF indicator is set and fgetc returns EOF. If a read error occurs, the error indicator is set and fgetc returns EOF. If an error occurs, `errno` is set.

Remarks

The fgetc function gets the next character from the file designated by `fp`. The character is signed.

See Also

fgetc Example

```
#include <stdio.h>

main ()
{
   FILE   *fp;
   int    c;
   fp = fopen ("data.fil", "r");
   while ( (c = fgetc (fp) ) != EOF)
      putchar (c);
```

```
    fclose (fp);
}
```

fgetchar

Equivalent to fgetc with the argument `stdin` (implemented for NetWare® 3.11 and above)

Local Servers: blocking

Remote Servers: blocking

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

int fgetchar  (void);
```

Return Values

The fgetchar function returns the next character from the input stream pointed to by `stdin`. If the stream is at end-of-file, the EOF indicator is set and fgetchar returns EOF. If a read error occurs, the error indicator is set and fgetchar returns EOF. When an error has occurred, the global variable `errno` contains a value indicating the type of error detected.

Remarks

The fgetchar function is equivalent to fgetc with the argument `stdin`.

See Also

fgetc (page 264), getc (page 297), getchar (page 299)

fgetchar Example

```
#include <stdio.h>

main()
{
   int c;
   while( (c = fgetchar()) != EOF )
      putchar(c);
}
```

fgetpos

Stores the current position of a stream in an object

Local Servers: blocking

Remote Servers: blocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

int fgetpos  (
   FILE     *fp,
   fpos_t   *pos);
```

Parameters

fp

(IN) Points to the file.

pos

(OUT) Points to the object into which the position of the file is stored.

Return Values

The fgetpos function returns a value of 0 if successful. Otherwise, the fgetpos function returns a nonzero value. If an error occurs, `errno` is set.

Remarks

The fgetpos function stores the current position of the file pointed to by `fp` in the object pointed to by `pos`. The value stored is usable by the fsetpos function for repositioning the file to the position it had at the time of the call to fgetpos.

See Also

fgetpos Example

```
#include <stdio.h>

main ()
{
```

```
        int    completionCode;
        FILE   *fp;
        fpos_t position;
        completionCode = fgetpos (fp, &position);
        completionCode = fsetpos (fp, &position);
}
```

fgets

Gets a string of characters from a stream and stores them in an array

Local Servers: blocking

Remote Servers: blocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

char *fgets  (
   char     *buf,
   size_t    n,
   FILE     *fp);
```

Parameters

buf

   (OUT) Points to the array into which the characters are to be stored.

n

   (IN) Specifies the number of characters to read.

fp

   (IN) Points to the file to be read.

Return Values

The fgets function returns `buf` if successful. NULL is returned if end-of-file is encountered or if a read error occurs. If an error occurs, `errno` is set.

If you link your application to the PRELUDE.OBJ file, the first character in the buffer is set to zero (0).

If you link your application to the NWPRE.OBJ file, the buffer passed to fgets is left untouched if the return is set to zero (0).

Remarks

The fgets function gets a string of characters from the file designated by `fp` and stores them in the array pointed to by `buf`. The fgets function stops reading characters when end-of-file is reached, or when a newline character is read, or when `n`-1 characters have been read, whichever comes first. The newline character is not discarded. A NULL character is placed immediately after the last character read into the array.

The gets function is similar to fgets except that it operates with stdin ; it has no size argument, and it replaces a newline character with the NULL character.

For backward compatibility with versions of CLIB.NLM previous to version 4.11, applications should link to PRELUDE.OBJ. For ANSI or POSIX compliance, applications should link to NWPRE.OBJ. Applications cannot link to both.

See Also

fgets Example

```
#include <stdio.h>

main ()
{
   FILE   *fp;
   char    buffer[80];
   fp = fopen ("data.fil", "r");
   while (fgets (buffer, 80, fp) != NULL)
      fputs (buffer, stdout);
   fclose (fp);
}
```

fileno

Returns the file handle for a stream (function or macro)

Local Servers: nonblocking

Remote Servers: nonblocking

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

int fileno  (
   FILE   *fp);
```

Parameters

fp

   (IN) Points to a stream opened with a previous call to fopen or fdopen.

Return Values

fileno returns the file handle designated by the `fp` parameter.

If an error occurs, `errno` is set to:

4	EBADF	Bad file number.

If fileno does not complete successfully, `NetWareErrno` is set.

Remarks

The returned file handle can be used to access the stream with any of the functions that take a handle.

There are two versions of fileno in the NetWare API:

- The fileno macro in STDIO.H does not do any error checking against a bad file pointer.
- The fileno function checks the handle passed in. If you want to use this function and are including STDIO.H, place a sequence such as the following in your code:

```
        #ifdef fileno
        #undef fileno
        #endif
```

See Also

close (page 205), creat (page 207), eof (page 213), filelength (page 216), fdopen (page 256), fopen (page 274), fstat (page 217), isatty (page 223), lseek (page 226), open (page 229), read (page 234), sopen (page 238), tell (page 242), write (page 246)

flushall

Clears all buffers associated with input streams and writes any buffers associated with output streams

Local Servers: blocking

Remote Servers: blocking

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

int flushall  (void);
```

Return Values

The flushall function returns the number of open streams. If an output error occurs while writing to a file, `errno` is set.

Remarks

The flushall function clears all buffers associated with input streams and writes any buffers associated with output streams. A subsequent read operation on an input file causes new data to be read from the associated file or device.

IMPORTANT: This function was designed to be used by applications that are cleaning up before exiting. If you need to flush all write buffers and then continue to run, you should call the the fflush function with the `fp` parameter set to NULL.

Novell recommends using LibC instead of CLib for any new NLM development. See Libraries for C (LibC) (http://developer.novell.com/ndk/libc.htm).

See Also

flushall Example

```
#include <stdio.h>

main ()
{
   printf ("The number of open files is %d\n", flushall () );
}
```

fopen

Opens a file and associates a stream with it

Local Servers: blocking

Remote Servers: blocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

FILE *fopen  (
   const char   *filename,
   const char   *mode);
```

Parameters

filename

(IN) Points to the name of the file to be opened.

mode

(IN) Points to the file mode.

Return Values

The fopen function returns a pointer to the object controlling the stream. This pointer must be passed as a parameter to subsequent functions for performing operations on the file. If the open operation fails, fopen returns NULL. If an error occurs, `errno` is set.

Remarks

This function also works on the DOS partition.

The fopen function opens the file whose name is the string pointed to by `filename`, and associates a stream with it. The argument `mode` points to a string beginning with one of the following sequences:

r	Opens file for reading; uses default file translation
w	Creates file for writing, or truncates to zero length; uses default file translation.
a	Appends; opens or creates text file for writing at end-of-file; uses default file translation.
rb	Opens binary file for reading.

rt	Opens text file for reading.
wb	Creates binary file for writing, or truncates to zero length.
wt	Creates text file for writing, or truncates to zero length.
ab	Appends; opens or creates binary file for writing at end-of-file.
at	Appends; opens or creates text file for writing at end-of-file.
r+	Opens file for update (reading and/or writing); uses default file translation.
w+	Creates file for update, or truncates to zero length; uses default file translation.
a+	Appends; opens or creates file for update, writing at end-of-file; uses default file translation.
r+b	Opens binary file for update (reading and/or writing).
r+t	Opens text file for update (reading and/or writing).
w+b	Creates binary file for update, or truncates to zero length.
w+t	Creates text file for update, or truncates to zero length.
a+b	Appends; opens or creates binary file for update, writing at end-of-file.
a+t	Appends; opens or creates text file for update, writing at end-of-file.
rb+	Opens binary file for update (reading and/or writing).
rt+	Opens text file for update (reading and/or writing).
wb+	Creates binary file for update, or truncates to zero length.
wt+	Creates text file for update, or truncates to zero length.
ab+	Appends; opens or creates binary file for update, writing at end-of-file.
at+	Appends; opens or creates text file for update, writing at end-of-file.

Opening a file with read mode (r as the first character in the `mode` argument) fails if the file does not exist or if it cannot be read. Opening a file with append mode (a as the first character in the `mode` argument) causes all subsequent writes to the file to be forced to the current end-of-file, regardless of previous calls to the fseek function. When a file is opened with update mode (+ as the second or third character of the `mode` argument), both input and output can be performed on the associated stream.

NOTE: For an example of how to reverse the effect of redirecting `stdin`, see the example for fdopen (page 256).

See Also

fclose (page 254), fcloseall (page 255), fdopen (page 256), freopen (page 285)

fopen Example

```
#include <stdio.h>
```

```
main ()
{
   char    filename[13];
   FILE    *fp;
   strcpy (filename, "REPORTAA.DAT");
   fp = fopen (filename, "r");

/* Do something */

   fclose (fp);
}
```

fprintf

Writes output to a stream under format control

Local Servers: blocking

Remote Servers: blocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

int fprintf  (
   FILE         *fp,
   const char   *format,
   ... );
```

Parameters

fp

(IN) Points to the file to be written to.

format

(IN) Points to the format control string.

Return Values

The fprintf function returns the number of characters written or a negative value if an output error occurred. If an error occurs, `errno` is set.

Remarks

The fprintf function writes output to the file pointed to by `fp` under control of the argument `format`. The format string is described under the description of the printf function.

See Also

fprintf Example

```
#include <stdio.h>

main ()
{
```

```
        char    *weekday = {"Saturday"};
        char    *month = {"April"};
        fprintf (stdout, "%s, %s %d, %d\n", weekday, month, 18, 1991);
}
```

produces the following:

```
Saturday, April 18, 1991
```

fputc

Writes a character to the output stream

Local Servers: blocking

Remote Servers: blocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

int fputc  (
   int    c,
   FILE   *fp);
```

Parameters

c

> (IN) Specifies the character to be written.

fp

> (IN) Points to the output stream.

Return Values

The fputc function returns the character written. If a write error occurs, the error indicator is set and fputc returns EOF. If an error occurs, `errno` is set.

Remarks

The fputc function writes the character specified by the argument `c` to the output stream designated by `fp`.

See Also

fclose (page 254), fgetc (page 264), fopen (page 274), fputs (page 281), putc (page 307), puts (page 311)

fputc Example

```
#include <stdio.h>

main ()
{
```

```
        FILE    *fp;
        int     c;
        fp = fopen ("data.fil", "r");
        while ( (c = fgetc (fp) ) != EOF)
           fputc (c, stdout);
        fclose (fp);
    }
```

fputs

Writes a character string to the output stream

Local Servers: blocking

Remote Servers: blocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

int fputs  (
   const char   *buf,
   FILE         *fp);
```

Parameters

buf

   (IN) Points to the character string to be written.

fp

   (IN) Points to the file

Return Values

If you have linked your program with NWPRE.OBJ, fputs returns the number of character put on success.

If you have linked your program with PRELUDE.OBJ, fputs returns -1 for failure and zero for success.

If an error occurs, `errno` is set.

Remarks

The fputs function writes the character string pointed to by `buf` to the file designated by `fp`. The terminating NULL character is not written.

For backward compatibility with versions of CLIB.NLM previous to version 4.11, applications should link to PRELUDE.OBJ. For ANSI or POSIX compliance, applications should link to NWPRE.OBJ. Applications cannot link to both.

See Also

fopen (page 274), fputc (page 279), putc (page 307), puts (page 311)

fputs Example

```
#include <stdio.h>

main ()
{
   FILE    *fp;
   char     buffer [80];
   fp = fopen ("data.fil", "r");
   while (fgets (buffer, 80, fp) ) != NULL)
       fputs (buffer, stdout);
   fclose (fp);
}
```

fread

Reads data from a stream

Local Servers: blocking

Remote Servers: blocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

size_t fread  (
   void      *buf,
   size_t    elsize,
   size_t    nelem,
   FILE      *fp);
```

Parameters

buf

   (OUT) Points to the location to receive data.

elsize

   (IN) Specifies the size (in bytes) of each element.

nelem

   (IN) Specifies the number of elements.

fp

   (IN) Points to the file to be read.

Return Values

The fread function returns the number of complete elements successfully read. This value can be less than the requested number of elements.

Call feof and ferror to determine whether the end of the file was encountered, or if an input/output error has occurred. If an error occurs, `errno` is set.

Remarks

This function also works on the DOS partition.

The fread function reads `nelem` elements of `elsize` bytes each from the file specified by `fp`.

See Also

fread Example

The following example reads a simple student record containing binary data. The student record is described by the struct student_data declaration.

```
#include <stdio.h>

struct student_data
{
   int student_id;
   unsigned char marks [10];
};

int   read_data (FILE *fp, struct student_data *p)
{
   return (fread (p, sizeof (*p), 1, fp) );
}
```

freopen

Opens a file and associates a stream with it

Local Servers: blocking

Remote Servers: blocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

FILE *freopen  (
   const char   *filename,
   const char   *mode,
   FILE         *fp);
```

Parameters

filename

(IN) Points to the name of the file to be opened.

mode

(IN) Points to the file mode.

fp

(IN) Points to the file structure.

Return Values

The freopen function returns a pointer to the object controlling the stream. This pointer must be passed as a parameter to subsequent functions for performing operations on the file. If the open operation fails, freopen returns NULL. If an error occurs, `errno` is set.

Remarks

The stream located by the `fp` pointer is closed. The freopen function opens the file whose name is the string pointed to by `filename`, and associates a stream with it. The stream information is placed in the structure located by the `fp` pointer.

The argument `mode` is described in the description of the fopen function.

NOTE: For an example of how to reverse the effect of redirecting `stdin`, see the example for fdopen (page 256).

See Also

freopen Example

```c
#include <stdio.h>
main ()

{
   FILE   *fp;
   fp = freopen ("report.dat", "r", stdin);
}
```

fscanf

Scans input from a stream under format control

Local Servers: blocking

Remote Servers: blocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

int fscanf  (
   FILE        *fp,
   const char  *format,
   ... );
```

Parameters

fp

   (IN) Points to the file.

format

   (IN) Points to the format control string.

Return Values

The fscanf function returns EOF when the scanning is terminated by reaching the end of the input stream. Otherwise, it returns the number of input arguments for which values were successfully scanned and stored. If a file input error occurs, `errno` is set.

Remarks

The fscanf function scans input from the file designated by `fp` under control of the argument `format`. Following the format string is a list of addresses to receive values. The format string is described under the description of the scanf function.

See Also

scanf (page 314), sscanf

fscanf Example

To scan a date in the form "Saturday April 18 1991":

```
#include <stdio.h>

main ()
{
   int    day, year;
   char   weekday[10], month[12];
   FILE   *in_data;
   in_data = fopen ("mydates.dat", "r");
   fscanf (in data, "%s %s %d %d", weekday, month, &day, &year);
}
```

fseek

Changes the read/write position of a stream

Local Servers: nonblocking

Remote Servers: nonblocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

int fseek  (
   FILE        *fp,
   long int    offset,
   int         where);
```

Parameters

fp

   (IN) Points to the file.

offset

   (IN) Specifies the file position to seek.

where

   (IN) Specifies the relative file position.

Return Values

The fseek function returns a value of 0 if successful, nonzero otherwise. If an error occurs, `errno` is set.

Remarks

The fseek function changes the read/write position of the file specified by `fp`. This position defines the character to be read or written on the next I/O operation on the file. The argument `fp` is a file pointer returned by fopen or freopen. The argument `offset` is the position to seek, relative to one of three positions specified by the argument `where`. Allowable values for the `where` parameter are:

SEEK_SET	Relative to beginning of file; the offset must be positive.
SEEK_CUR	Relative to the current position in the file.
SEEK_END	Relative to the end of the file.

The fseek function clears the end-of-file indicator and undoes any effects of the ungetc function on the same file.

Call ftell to obtain the current position in the file before changing it. Restore the position by using the value returned by ftell in a subsequent call to fseek with the `where` parameter set to SEEK_SET.

See Also

fgetpos (page 267), fopen (page 274), fsetpos (page 291), ftell (page 293), lseek (page 226)

fseek Example

You can determine the size of a file by means of the following example, which saves and restores the current position of the file.

```
#include <stdio.h>

long int   filesize (FILE *fp)
{
   long int   save_pos, size_of_file;
   save_pos = ftell (fp);
   fseek (fp, 0L, SEEK_END);
   size_of_file = ftell (fp);
   fseek (fp, save_pos, SEEK_SET);
   return (size_of_file);
}
```

fsetpos

Positions a stream according to the value of an object

Local Servers: nonblocking

Remote Servers: nonblocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

int fsetpos  (
   FILE          *fp,
   const fpos_t  *pos);
```

Parameters

fp

   (IN) Points to the file.

pos

   (IN) Points to the object that specifies the new file position.

Return Values

The fsetpos function returns a value of 0 if successful; otherwise, the fsetpos function returns a nonzero value. If an error occurs, `errno` is set.

Remarks

The fsetpos function positions the file pointed to by `fp` according to the value of the object pointed to by `pos`, which shall be a value returned by an earlier call to the fgetpos function on the same file.

See Also

fsetpos Example

```
#include <stdio.h>

main ()
{
   FILE      *fp;
```

```
        fpos_t    position;
        fgetpos (fp, &position);
        fsetpos (fp, &position);
}
```

ftell

Returns the current read/write position of a stream

Local Servers: nonblocking

Remote Servers: nonblocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

long int ftell  (
   FILE   *fp);
```

Parameters

fp

> (IN) Points to the file.

Return Values

The ftell function returns the current read/write position of the file specified by `fp`. When an error is detected, a value of -1 is returned. If an error occurs, `errno` is set.

Remarks

The ftell function returns the current read/write position of the file specified by `fp`. This position defines the character to be read or written by the next I/O operation on the file. You can use the value returned by ftell in a subsequent call to fseek in order to set the file to the same position.

See Also

ftell Example

You can determine the size of a file by using the following example, which saves and restores the current position of the file.

```
#include <stdio.h>

long int   filesize (FILE *fp)

{
```

```
        long int    save_pos, size_of_file;
        save_pos = ftell (fp);
        fseek ( fp, 0L, SEEK_END);
        size_of_file = ftell (fp);
        fseek (fp, save_pos, SEEK_SET);
        return (size_of_file);
}
```

fwrite

Writes elements to a stream

Local Servers: blocking

Remote Servers: blocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

size_t fwrite  (
   const void   *buf,
   size_t        elsize,
   size_t        nelem,
   FILE         *fp);
```

Parameters

buf

   (IN) Points to the buffer containing the data to write.

elsize

   (IN) Specifies the size (in bytes) of each element.

nelem

   (IN) Specifies the number of elements.

fp

   (IN) Points to the file.

Return Values

This function also works on the DOS partition.

The fwrite function returns the number of complete elements that are successfully written. This value is less than the requested number of elements only if a write error occurs.

Remarks

The fwrite function writes `nelem` elements of `elsize` bytes each to the file specified by `fp`.

See Also

fwrite Example

The following example writes a simple student record containing binary data.

```c
#include <stdio.h>

struct student_data
{
   int            student_id;
   unsigned char  marks[10];
};

int write_data (FILE *fp, struct student_data *p)
{
   return (fwrite ( p, sizeof (*p), 1, fp) );
}
```

getc

Gets the next character from a stream (function or macro)

Local Servers: blocking

Remote Servers: blocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

int getc  (
   FILE   *fp);
```

Parameters

fp

   (IN) Points to the file.

Return Values

The getc function or macro returns the next character from the input stream pointed to by `fp`. If the stream is at end-of-file, the EOF indicator is set and getc returns EOF. If a read error occurs, the error indicator is set and getc returns EOF. If an error occurs, `errno` is set.

Remarks

The getc function or macro gets the next character from the file designated by `fp`. The character is returned as an int value.

The getc function is equivalent to fgetc, except that it can be implemented as a macro.

See Also

getc Example

```
#include <stdio.h>

main ()
{
   FILE     *fp;
   int       c;
```

```
        fp = fopen ("data.fil", "r");
        while ( (c = getc (fp) ) != EOF)
           putchar (c);
        fclose (fp);
    }
```

getchar

Equivalent to getc with the argument `stdin` (function or macro)

Local Servers: blocking

Remote Servers: blocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

int getchar  (void);
```

Return Values

The getchar function or macro returns the next character from the input stream pointed to by `stdin`. If the stream is at end-of-file, the EOF indicator is set and getchar returns EOF. If a read error occurs, the error indicator is set and getchar returns EOF. If an error occurs, `errno` is set.

Remarks

The getchar function or macro is equivalent to getc with the argument `stdin`.

See Also

getchar Example

```
#include <stdio.h>

main ()
{
   FILE    *fp;
   int      c;
   fp = freopen ("data.fil", "r", stdin);
   while ( (c = getchar () ) != EOF)
      putchar (c);
   fclose (fp);
}
```

gets

Gets a string of characters from `stdin` and stores them in an array

Local Servers: blocking

Remote Servers: blocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

char *gets  (
   char    *buf);
```

Parameters

buf

 (OUT) Points to the array into which the characters are to be stored.

Return Values

The gets function returns `buf` if successful. It returns NULL if a read error occurs.

Remarks

The gets function gets a string of characters from the file designated by `stdin` and stores them in the array pointed to by `buf` until a newline character is read. Any newline character is discarded, and a NULL character is placed immediately after the last character read into the array.

It is recommended that fgets be used instead of gets because data beyond the array `buf` is destroyed if a newline character is not read from the input stream `stdin` before the end of the array `buf` is reached.

See Also

fgetc (page 264), fgets (page 269), fopen (page 274), getc (page 297), ungetc (page 324)

gets Example

```
#include <stdio.h>

main ()
{
   char    buffer[80];
```

```
    while (gets (buffer) ) != NULL)
        puts (buffer);
}
```

printf

Writes formatted output to a specified file designated by `stdout`

Local Servers: blocking

Remote Servers: blocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

int printf  (
   const char   *format,
   ... );
```

Parameters

format

   (IN) Points to the format control string.

Return Values

The printf function returns the number of characters written, or it returns a negative value if an output error occurred. If an error occurs, `errno` is set.

Remarks

The printf function writes output to the file designated by `stdout` under control of the argument `format.`

The format control string consists of ordinary characters, which are written exactly as they occur in the format string, and of conversion specifiers, which cause argument values to be written as they are encountered during the processing of the format string. An ordinary character in the format string is any character, other than a percent (%) character, that is not part of a conversion specifier. A conversion specifier is a sequence of characters in the format string. The conversion specifier begins with a % and is followed, in sequence, by the following:

- Zero or more format control flags, which can modify the final effect of the format directive.
- An optional decimal integer, or an asterisk character (*), which specifies a minimum field width to be reserved for the formatted item.
- An optional precision specification in the form of a period character (.) followed by an optional decimal integer or an asterisk character (*).
- An optional type-length specification. It can be any one of the following characters:

   h l L N F

- A character that specifies the type of conversion to be performed. It can be any one of the following characters:

  c d e E f F g G i n o p s u x X

The valid format control flags are:

-	The formatted item is left-justified within the field; normally, items are right-justified.
	A signed, positive object always starts with a plus (+) character; normally, only negative items begin with a sign.
" "	A signed, positive object always starts with a space character; if both + and - are specified, + overrides -.
#	An alternate conversion form is used:
	For o (unsigned octal) conversions, the precision is incremented so that the first digit is 0.
	For x or X (unsigned hexadecimal) conversions, a nonzero value is prepended with 0x or 0X, respectively.
	For e, E, f, g, or G (any floating-point) conversions, the result always contains a decimal-point character, even if no digits follow it; normally, a decimal- point character appears in the result only if there is a digit to follow it.
	In addition to the preceding, for g or G conversions, trailing zeros are not removed from the result.

If no field width is specified, or if the value that is given is less than the number of characters in the converted value (subject to any precision value), a field of sufficient width to contain the converted value is used. If the converted value has fewer characters than are specified by the field width, the value is padded on the left (or right, subject to the left-justification flag) with spaces or zero characters (0). If the field width begins with a zero, the value is padded with zeros; otherwise, the value is padded with spaces. If the field width is *, a value of type int from the argument list is used (before a precision argument or a conversion argument) as the minimum field width. A negative field width value is interpreted as a left- justification flag, followed by a positive field width.

As with the field width specifier, a precision specifier of * causes a value of type int from the argument list to be used as the precision specifier. If no precision value is given, a precision of 0 is used. The precision value affects the following conversions:

- For *d*, *i*, *o*, *u*, *x*, and *X* (integer) conversions, the precision specifies the minimum number of digits to appear.
- For *e*, *E*, and *f* (fixed-precision, floating-point) conversions, the precision specifies the number of digits to appear after the decimal-point character.
- For *g* and *G* (variable-precision, floating-point) conversions, the precision specifies the maximum number of significant digits to appear.
- For *s* (string) conversions, the precision specifies the maximum number of characters to appear.

A type length specifier affects the conversion as follows:

---

h	Causes a d, i, o, u, x, or X (integer) conversion to process a short int or unsigned short int argument; note that although the argument can have been promoted as part of the function call, the value is converted to the smaller type before it is converted.

	It causes an n (converted length assignment) operation to assign the converted length to an object of type unsigned short int.
l	Causes a d, i, o, u, x, or X (integer) conversion to process a long int or unsigned long int argument.
	It causes an n (converted length assignment) operation to assign the converted length to an object of type unsigned long int.
F	Causes the pointer associated with n, p, or s conversions to be treated as a far pointer.
L	Causes an e, E, f, g, or G (double) conversion to process a long double argument.
N	Causes the pointer associated with n, p, or s conversions to be treated as a near pointer.

The valid conversion type specifiers are:

c	An argument of type int is converted to a value of type char and the corresponding ASCII character code is written to the output stream.
d, i	An argument of type int is converted to a signed decimal notation and written to the output stream. The default precision is 1, but if more digits are required, leading zeros are added.
e, E	An argument of type double is converted to a decimal notation in the form [-]d.ddde[+|-]ddd similar to FORTRAN exponential (**E**) notation. The leading sign appears (subject to the format control flags) only if the argument is negative. If the argument is nonzero, the digit before the decimal point character is nonzero. The precision is used as the number of digits following the decimal point character. If the precision is not specified, a default precision of 6 is used. If the precision is 0, the decimal point character is suppressed. The value is rounded to the appropriate number of digits. For **E** conversions, the exponent begins with the character **E** rather than **e**. The exponent sign and a three-digit number (that indicates the power of ten by which the decimal fraction is multiplied) are always produced.
f	An argument of type double is converted to a decimal notation in the form [-]ddd.ddd similar to FORTRAN fixed-point (**F**) notation. The leading sign appears (subject to the format control flags) only if the argument is negative. The precision is used as the number of digits following the decimal point character. If the precision is not specified, a default precision of 6 is used. If the precision is 0, the decimal point character is suppressed, otherwise, at least one digit is produced before the decimal point character. The value is rounded to the appropriate number of digits.
g, G	An argument of type double is converted using either the **f** or e (or **E**, for a **G** conversion) style of conversion depending on the value of the argument. In either case, the precision specifies the number of significant digits that are contained in the result. The **e** style conversion is used only if the exponent from such a conversion would be less than -4 or greater than the precision. Trailing zeros are removed from the result and a decimal-point character only appears if it is followed by a digit.
n	The number of characters that have been written to the output stream is assigned to the integer pointed to by the argument. No output is produced.
o	An argument of type int is converted to an unsigned octal notation and written to the output stream. The default precision is 1, but if more digits are required, leading zeros are added.
p, P	An argument of type void * is converted to a value of type int and the value is formatted as for a hexadecimal (**x**) conversion.
s	Characters from the string specified by an argument of type char *, up to, but not including, the terminating NULL character (\0), are written to the output stream. If a precision is specified, no more than that many characters are written.

S	Characters from a length-preceded string are written to the output stream. If a precision is specified, no more than that many characters are written.
u	An argument of type int is converted to an unsigned decimal notation and written to the output stream. The default precision is 1, but if more digits are required, leading zeros are added.
x, X	An argument of type int is converted to an unsigned hexadecimal notation and written to the output stream. The default precision is 1, but if more digits are required, leading zeros are added. Hexadecimal notation uses digits (0 through 9) and characters (a through f or A through F) for **x** or **X** conversions respectively, as the hexadecimal digits. Subject to the alternate-form control flag, **0x** or **0X** is affixed to the output.

Any other conversion type specifier character, including another percent (*%*) character, is written to the output stream with no special interpretation.

The arguments must correspond with the conversion type specifiers, left to right in the string; otherwise, indeterminate results occur.

For example, a specifier of the form %8.*f defines a field to be at least 8 characters wide and gets the next argument for the precision to be used in the conversion.

The output from

```
printf ("f1 = %8.4f f2 = %10.2E x = %#08x i = %d",
       23.45,   3141.5926,   0x1db,    -1 );
```

would be

```
f1 =  23.4500 f2 =   3.14E+003 x = 0x0001db i = -1
```

You can also use strings similar to the following:

```
printf ("Test: %3$s %2$d %1$s", string, 10, string);
```

See Also

printf Example

```
#include <stdio.h>

main ()
{
   char    *weekday, *month;
   int     day, year;
   weekday = "Saturday";
   month = "April";
   day = 18;
   year = 1991;
   printf ("%s, %s %d, %d\n", weekday, month, day, year);
}
```

produces the following:

Saturday, April 18, 1991

putc

Writes a character to the output stream (function or macro)

Local Servers: blocking

Remote Servers: blocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

int putc  (
   int     c,
   FILE    *fp);
```

Parameters

c

> (IN) Specifies the character to be written.

fp

> (IN) Points to the file.

Return Values

The putc function or macro returns the character written. If a write error occurs, the error indicator is set and putc returns EOF.

Remarks

The putc function is equivalent to fputc, except that it can be implemented as a macro. The putc function or macro writes the character specified by the argument c to the output stream designated by fp.

See Also

ferror (page 261), fopen (page 274), fputs (page 281), puts (page 311)

putc Example

```
#include <stdio.h>

main ()
{
```

```
        FILE    *fp;
        int    c;
        fp = fopen ("data.fil", "r");
        while ( (c = fgetc( fp )) != EOF)
            putc (c, stdout);
        fclose (fp);
}
```

putchar

Writes a character to the output stream (function or macro)

Local Servers: blocking

Remote Servers: blocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

int putchar  (
   int   c);
```

Parameters

c

(IN) Specifies the character to be written.

Return Values

This function or macro returns the character written. If a write error occurs, the error indicator is set and putchar returns EOF. If an error occurs, `errno` is set.

Remarks

The putchar function or macro writes the character specified by the argument `c` to the output stream `stdout`.

The function is equivalent to:

```
   fputc (c, stdout);
```

See Also

putchar Example

```
#include <stdio.h>

main ()
{
   FILE    *fp;
   int      c;
```

```
      fp = fopen ("data.fil", "r");
      c = fgetc (fp);
      while (c != EOF)
      {
         putchar (c);
         c = fgetc (fp);
      };
      fclose (fp);
}
```

puts

Writes a specified character string to the output stream and appends a newline character to the output

Local Servers: blocking

Remote Servers: blocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

int puts  (
   const char   *buf);
```

Parameters

buf

   (IN) Points to the character string.

Return Values

The puts function returns a nonzero value if an error occurs; otherwise, it returns a value of 0. If an error occurs, `errno` is set.

Remarks

The puts function writes the character string pointed to by `buf` to the output stream designated by `stdout` and appends a newline character to the output. The terminating NULL character is not written.

See Also

puts Example

```
#include <stdio.h>

main ()
{
   FILE   *fp;
   char    buffer[80];
   fp = freopen ("data.fil", "r", stdin);
   while (gets (buffer) != NULL)
```

```
      puts (buffer);
   fclose (fp);
}
```

rewind

Sets the file position indicator to the beginning of the file

Local Servers: blocking

Remote Servers: blocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

void rewind  (
   FILE   *fp);
```

Parameters

fp

   (IN) Points to the file.

Remarks

The rewind function sets the file position indicator for the stream indicated by fp to the beginning of the file. It is equivalent to:

```
   fseek( fp, 0L, SEEK_SET );
```

except that the error indicator for the stream is cleared.

See Also

clearerr (page 253), fopen (page 274)

rewind Example

```
#include <stdio.h>

FILE    *fp;

if ( (fp = fopen ("program.asm", "r") ) != NULL)
{
   assemble_pass ( 1 );
   rewind (fp);
   assemble_pass ( 2 );
   fclose (fp);
}
```

scanf

Scans input from a stream

Local Servers: blocking

Remote Servers: blocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

int scanf  (
   const char   *format,
   ... );
```

Parameters

format

   (IN) Points to the format control string.

Return Values

The scanf function returns EOF when the scanning is terminated by reaching the end of the input stream. Otherwise, the number of input arguments for which values were successfully scanned and stored is returned.

Remarks

The scanf function scans input from the file designated by stdin under control of the argument format. Following the format string is the list of addresses of items to receive values.

The format control string consists of zero or more format directives that specify acceptable input file data. Subsequent arguments are pointers to various types of objects that are assigned values as the format string is processed.

A format directive can be a sequence of one or more white-space characters, an ordinary character, or a conversion specifier. An ordinary character in the format string is any character, other than a white-space character or the percent (%) character, that is not part of a conversion specifier. A conversion specifier is a sequence of characters in the format string, which begins with a % and is followed, in sequence, by the following:

- An optional assignment suppression indicator: the asterisk character (*)
- An optional decimal integer that specifies the maximum field width to be scanned for the conversion
- An optional pointer type specification: one of *N* or *F*

- An optional type length specification: one of *h*, *l* or *L*
- A character that specifies the type of conversion to be performed. It can be any one of the following characters: c, d, e, f, g, i, o, n, p, s, u, x, or [.

As each format directive in the format string is processed, the directive can successfully complete, fail because of a lack of input data, or fail because of a matching error as defined by the particular directive. If end-of-file is encountered on the input data before any characters that match the current directive have been processed (other than leading white-space where permitted), the directive fails for lack of data. If end-of-file occurs after a matching character has been processed, the directive is completed (unless a matching error occurs), and the function returns without processing the next directive. If a directive fails because of an input character mismatch, the character is left unread in the input stream. Trailing white-space characters, including newline characters, are not read unless matched by a directive. When a format directive fails, or the end of the format string is encountered, the scanning is completed and the function returns.

When one or more white-space characters—space, horizontal tab (\t), vertical tab (\v), form feed (\f), carriage return (\r), newline or line feed (\n)—occur in the format string, input data up to the first nonwhite-space character is read, or until no more data remains. If no white-space characters are found in the input data, the scanning is complete and the function returns.

An ordinary character in the format string is expected to match the same character in the input stream.

A conversion specifier in the format string is processed as follows:

- For conversion types other than [, *c*, and *n*, leading white-space characters are skipped.
- For conversion types other than n, all input characters, up to any specified maximum field length, that can be matched by the conversion type are read and converted to the appropriate type of value; the character immediately following the last character to be matched is left unread; if no characters are matched, the format directive fails.
- Unless the assignment suppression indicator (*) was specified, the result of the conversion is assigned to the object pointed to by the next unused argument (if assignment suppression was specified, no argument is skipped); the arguments must correspond in number, type, and order to the conversion specifiers in the format string.

A pointer type specification is used to indicate the type of pointer used to locate the next argument to be scanned:

F	Points to a far pointer.
N	Points to a near pointer.

The pointer type defaults to that used for data in the memory model for which the program has been compiled.

A type length specifier affects the conversion as follows:

h	Causes a **d**, **i**, **o**, **u**, or **x** (integer) conversion to assign the converted value to an object of type short int or unsigned short int.
	It causes an **n** (read length assignment) operation to assign the number of characters that have been read to an object of type unsigned short int.

l	Causes a **d**, **i**, **o**, **u**, or **x** (integer) conversion to assign the converted value to an object of type long int or unsigned long int.
	It causes an **n** (read length assignment) operation to assign the number of characters that have been read to an object of type unsigned long int.
	It causes an **e**, **f**, or **g** (floating point) conversion to assign the converted value to an object of type double.
L	Causes an **e**, **f**, or **g** (floating point) conversion to assign the converted value to an object of type long double.

The valid conversion type specifiers are:

c	Any sequence of characters in the input stream of the length specified by the field width, or a single character if no field width is specified, is matched. The argument is assumed to point to the first element of a character array of sufficient size to contain the sequence, without a terminating NULL character (\0). For a single-character assignment, a pointer to a single object of type char is sufficient.
d	A decimal integer, consisting of an optional sign, followed by one or more decimal digits, is matched. The argument is assumed to point to an object of type int.
e, **f**, **g**	A floating-point number, consisting of an optional sign (+ or -), followed by one or more decimal digits, optionally containing a decimal-point character, followed by an optional exponent of the form **e** or **E**, an optional sign, and one or more decimal digits, is matched. The exponent, if present, specifies the power of ten by which the decimal fraction is multiplied. The argument is assumed to point to an object of type float.
i	An optional sign, followed by an octal, decimal, or hexadecimal constant is matched. An octal constant consists of zero and zero or more octal digits. A decimal constant consists of a nonzero decimal digit and zero or more decimal digits. A hexadecimal constant consists of the characters **0x** or 0X followed by one or more (upper- or lowercase) hexadecimal digits. The argument is assumed to point to an object of type int.
n	No input data is processed. Instead, the number of characters that have already been read is assigned to the object of type unsigned int that is pointed to by the argument. The number of items that have been scanned and assigned (the return value) is not affected by the n conversion type specifier.
o	An octal integer, consisting of an optional sign, followed by one or more (zero or nonzero) octal digits, is matched. The argument is assumed to point to an object of type int.
p	A hexadecimal integer, as described for **x** conversions below, is matched. The converted value is further converted to a value of type void* and then assigned to the object pointed to by the argument.
s	A sequence of nonwhite-space characters is matched. The argument is assumed to point to the first element of a character array of sufficient size to contain the sequence and a terminating NULL character, which is added by the conversion operation.
u	An unsigned decimal integer, consisting of one or more decimal digits, is matched. The argument is assumed to point to an object of type unsigned int.
x	A hexadecimal integer, consisting of an optional sign, followed by an optional prefix 0x or 0X, followed by one or more (uppercase or lowercase) hexadecimal digits, is matched. The argument is assumed to point to an object of type int.

[c1c2. ..]	A sequence of characters, consisting of any of the characters c1, **c2**, ... called the scanset, in any order, is matched. **c1** cannot be the caret character (^). If **c1** is], that character is considered to be part of the scanset and a second] is required to end the format directive. The argument is assumed to point to the first element of a character array of sufficient size to contain the sequence and a terminating NULL character, which is added by the conversion operation.
[^c1c2. ..]	A sequence of characters, consisting of any of the characters other than the characters between the ^ and], is matched. As with the preceding conversion, if **c1** is], it is considered to be part of the scanset and a second] ends the format directive. The argument is assumed to point to the first element of a character array of sufficient size to contain the sequence and a terminating NULL character, which is added by the conversion operation.

A conversion type specifier of % is treated as a single ordinary character that matches a single % character in the input data. A conversion type specifier other than those listed above causes scanning to terminate the function to return.

The line

```
scanf ("%s%*f%3hx%d", name, &hexnum, &decnum)
```

with input

```
some_string 34.555e-3 abc1234
```

copies some_string into the array name, skip 34.555e-3, assign 0xabc to hexnum and 1234 to decnum. The return value is 3.

The line

```
char fmt[100];
   strcpy (fmt, "%[abcdefghijklmnopqrstuvwxyz");
   strcat (fmt,"[ABCDEFGHIJKLMNOPQRSTUVWZ]%*2s%[W\n]");
   scanf (fmt, string1, string2)
```

with input

```
They may look alike, but they don't perform alike.
```

assigns

```
"They may look alike"
```

to string1, skip the comma and the space, and assign

```
" but they don't perform alike.".
```

to string2. (The %*2s only matches the ","; the next blank terminates that field.)

See Also

fscanf (page 287), sscanf (*NDK: Program Management*)

scanf Example

To scan a date in the form "Saturday April 18 1991":

```
#include  <stdio.h>
int day, year;
char weekday[10], month[12];
scanf ("%s %s %d %d", weekday, month, &day, &year);
```

setbuf

Associates a buffer with a file after the file is open and before it has been read or written to

Local Servers: nonblocking

Remote Servers: nonblocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

void setbuf  (
   FILE   *fp,
   char   *buffer);
```

Parameters

fp

   (IN) Points to the file.

buffer

   (IN) Points to the buffer.

Return Values

The setbuf function returns no value.

Remarks

The setbuf function can be used to associate a buffer with the file designated by `fp`. If this function is used, it must be called after the file has been opened and before it has been read or written. If the argument buffer is NULL, then all input/ output for the file pointed to by `fp` is completely unbuffered. If the argument buffer is not NULL, then it must point to an array that is at least BUFSIZ characters in length, and all input/output is fully buffered. BUFSIZ is a constant defined in STDIO.H.

See Also

fopen (page 274), setvbuf (page 321)

setbuf Example

```
#include <stdio.h>
```

```
main ()
{
    char    *buffer;
    FILE    *fp;
    fp = fopen ("data.fil", "r");
    buffer = malloc (BUFSIZ);
    setbuf (fp, buffer);
    fclose(fp);
}
```

setvbuf

Associates a buffer with a file after the file is open and before it has been read or written to

Local Servers: nonblocking

Remote Servers: nonblocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

int setvbuf  (
   FILE      *fp,
   char      *buf,
   int        mode,
   size_t     size);
```

Parameters

fp

   (IN) Points to the file.

buf

   (IN) Points to the buffer.

mode

   (IN) Specifies the file mode that determines how to buffer the file.

size

   (IN) Specifies the size of the array.

Return Values

The setvbuf function returns a value of 0 on success, or a nonzero value if an invalid value is given for `mode` or `size`.

Remarks

The setvbuf function can be used to associate a buffer with the file designated by `fp`. If this function is used, it must be called after the file has been opened and before it has been read or written. The argument `mode` determines how the file pointed to by `fp` is to be buffered, as follows:

_IOFBF	Causes input/output to be fully buffered.

_IOLBF	Causes output to be line buffered (the buffer is flushed when a newline character is written, when the buffer is full, or when input is requested).
_IONBF	Causes input/output to be completely unbuffered.

If the argument `buf` is not NULL, the array to which it points is used instead of an automatically allocated buffer. The argument `size` specifies the size of the array.

See Also

setvbuf Example

```
#include <stdio.h>

main ()
{
   char    *buf;
   FILE    *fp;
   fp = fopen ("data.fil", "r");
   buf = malloc (1024);
   setvbuf (fp, buf, _IOFBF, 1024);
   fclose(fp);
}
```

tmpfile

Creates a temporary binary file

Local Servers: blocking

Remote Servers: blocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

FILE *tmpfile  (void);
```

Return Values

The tmpfile function returns a pointer to the stream of the file that it created. If the file cannot be created, the tmpfile function returns NULL. If an error occurs, `errno` is set.

Remarks

The tmpfile function creates a temporary binary file that is automatically removed when it is closed or at program termination. The file is opened for update.

See Also

fopen (page 274), freopen (page 285), tmpnam (Multiple and Inter-File Services)

ungetc

Pushes a character back onto the specified input stream

Local Servers: nonblocking

Remote Servers: nonblocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>

int ungetc  (
   int     c,
   FILE    *fp);
```

Parameters

c

> (IN) Specifies the character to be pushed back onto the specified input stream.

fp

> (IN) Points to the input stream.

Return Values

The ungetc function returns the character pushed back.

Remarks

The ungetc function pushes the character specified by `c` back onto the input stream specified by `fp`. This character is returned by the next read from the stream. Only the last character returned in this way is remembered.

The ungetc function clears the EOF indicator, unless the value of `c` is EOF.

See Also

vfprintf

Writes output to a stream under format control

Local Servers: blocking

Remote Servers: blocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdarg.h>
#include <stdio.h>

int  vfprintf  (
   FILE         *fp,
   const char   *format,
   va_list       arg);
```

Parameters

fp

   (IN) Points to the file.

format

   (IN) Points to the format control string.

arg

   (IN) Specifies a variable argument.

Return Values

The vfprintf function returns the number of characters written or a negative value if an output error occurred. If an error occurs, `errno` is set.

Remarks

The vfprintf function writes output to the file pointed to by `fp` under control of the argument `format`. The format string is described under the description for printf. The vfprintf function is equivalent to fprintf, with the variable argument list replaced with `arg`, which has been initialized by the va_start macro.

See Also

fprintf (page 277), printf (page 302), sprintf, va_arg, va_end, va_start (*NDK: Program Management*)

vfprintf Example

```
#include <stdarg.h>
#include <stdio.h>

extern FILE *LogFile;

void errmsg          /* A GENERAL ERROR ROUTINE */
(char *format, ... )
{
   va_list arglist;
   va_start (arglist, format);
   vfprintf (stderr, format, arglist);
   va_end  (arglist );
   if  (LogFile != NULL)
   {
      va_start (arglist, format);
      vfprintf (LogFile, format, arglist);
      va_end (arglist);
   }
}
```

vfscanf

Scans input from a stream under format control

Local Servers: blocking

Remote Servers: blocking

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>
#include <stdarg.h>

int  vfscanf  (
   FILE         *fp,
   const char   *format,
   va_list       arg);
```

Parameters

fp

   (IN) Points to the file.

format

   (IN) Points to the format control string.

arg

   (OUT) Specifies a variable argument.

Return Values

The vfscanf function returns EOF when the scanning is terminated by reaching the end of the input stream. Otherwise, the number of input arguments for which values were successfully scanned and stored is returned. If a file input error occurs, `errno` is set.

Remarks

The vfscanf function scans input from the file designated by `fp` under control of the argument `format`. The format list is described with the scanf function.

The vfscanf function is equivalent to the fscanf function, with a variable argument list replaced with `arg`, which has been initialized using the va_start macro.

See Also

fscanf (page 287), scanf (page 314), sscanf, va_arg, va_end, va_start (*NDK: Program Management*)

vfscanf Example

```
#include <stdio.h>

#include <stdarg.h>
main ()
{
   auto va_list arglist;
   va_start (arglist, arg);
   vfprintf (fp, format, arglist);
   va_end (arglist);
}
```

vprintf

Writes output to `stdout` under format control

Local Servers: blocking

Remote Servers: blocking

Classification: ANSI

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdarg.h>
#include <stdio.h>

int  vprintf  (
   const char   *format,
   va_list        arg);
```

Parameters

format

   (IN) Points to the format control string.

arg

   (IN) Specifies a variable argument.

Return Values

The vprintf function returns the number of characters written or a negative value if an output error occurred. If an error occurs, `errno` is set.

Remarks

The vprintf function writes output to the file `stdout` under control of the argument `format`. The format string is described under the description for printf. The vprintf function is equivalent to printf, with the variable argument list replaced with `arg`, which has been initialized by the va_start macro.

See Also

fprintf (page 277), printf (page 302), sprintf, va_arg, va_end, va_start (*NDK: Program Management*)

vprintf Example

The following shows the use of vprintf in a general error message routine.

```
#include <stdarg.h>
#include <stdio.h>

void errmsg (char *format, ... )
{
   va_list arglist;
   printf ("Error: ");
   va_start (arglist, format);
   vprintf (format, arglist);
   va_end (arglist);
}
```

vscanf

Scans input from the stream designated by `stdin`

Local Servers: blocking

Remote Servers: blocking

Platform: NLM

Service: Stream I/O

Syntax

```
#include <stdio.h>
#include <stdarg.h>

int  vscanf  (
   const char   *format,
   va_list       arg);
```

Parameters

format

   (IN) Points to the format control string.

arg

   (OUT) Specifies a variable argument.

Return Values

The vscanf function returns EOF when the scanning is terminated by reaching the end of the input stream. Otherwise, the number of input arguments for which values were successfully scanned and stored is returned.

Remarks

The vscanf function scans input from the file designated by `stdin` under control of the argument `format`. The format list is described with the scanf function.

The vscanf function is equivalent to the scanf function, with a variable argument list replaced with `arg`, which has been initialized using the va_start macro

See Also

fscanf (page 287), scanf (page 314), sscanf, va_arg, va_end, va_start (*NDK: Program Management*)

vscanf Example

```
#include <stdio.h>
#include <stdarg.h>
```

```
void find (char *format, char *arg, ... )
{
   va_list arglist;
   va_start (arglist, arg);
   vscanf (format, arglist);
   va_end (arglist );
}
```

Synchronization Concepts

<div style="text-align: right; font-size: 3em;">20</div>

This documentation describes Synchronization, its functions, and features.

Synchronization provides developers with the ability to lock users out of a file while it is being accessed by someone else. Synchronization is essential to assuring data integrity on the network, where many users can access the same data simultaneously. Data locks are the basis for controlling file access.

NetWare® also supports semaphores for controlling access to files. However, semaphores can be applied to other resources as well, and so aren't exclusively a file synchronization mechanism.

In addition to locking data, you can scan for information about locks and semaphores such as a list of locks associated with a specified connection or a list of connections locking a specified file.

NOTE: NetWare 3.11 introduced numerous new NCP requests for file locking and semaphore management. Synchronization takes advantage of the new requests whenever possible.

20.1 Data Locks

NetWare® supports three types of data locks:

- "File Locks" on page 333 and "File Locking Functions" on page 333
- "Physical Record Locks" on page 334 and "Physical Record Locking Functions" on page 334
- "Logical Record Locks" on page 334 and "Logical Record Locking Functions" on page 334

20.1.1 File Locks

File locks control access to an entire file or several files at the same time. Once locked, a file can't be accessed by another connection.

20.1.2 File Locking Functions

These functions manage file locks:

NWClearFileLock2	Unlocks the specified file and removes it from the log table.
NWClearFileLockSet	Unlocks and removes all files logged in the log table.
NWLockFileLockSet	Locks all files logged in the log table.
NWLogFileLock2	Logs the specified file in the log table.
NWReleaseFileLock2	Unlocks the specified file but doesn't remove it from the log table.
NWReleaseFileLockSet	Unlocks all logged files but doesn't remove them from the log table.

20.1.3 Physical Record Locks

Physical record locks control access to byte ranges within a file. To lock a physical record, specify a starting offset within the file and the length of the record in bytes. Only the byte range is locked; the rest of the file remains free. Physical record locks can be exclusive or shareable.

20.1.4 Physical Record Locking Functions

These functions manage physical record locks:

NWClearPhysicalRecord	Unlocks the specified physical record and removes it from the log table.
NWClearPhysicalRecordSet	Unlocks all logged physical records and removes them from the log table.
NWLockPhysicalRecordSet	Unlocks and removes all physical records from the log table.
NWLogPhysicalRecord	Logs a physical record in the log table.
NWReleasePhysicalRecord	Unlocks the specified physical record but doesn't remove it from the log table.
NWReleasePhysicalRecordSet	Unlocks all logged physical records but doesn't remove them from the log table.

20.1.5 Logical Record Locks

Logical record locks control access to a logical record name. You define logical record names for the purposes of your application. By associating the logical record with a specific file or physical record you can coordinate access to data within your application.

The NetWare® server only reports the status of the logical record. It's up to your application to enforce restrictions implied by a logical record. The server doesn't prevent applications from accessing physical data you have associated with the logical record. For this reason, we recommend you don't rely on file or physical record locks when using logical record locks.

20.1.6 Logical Record Locking Functions

These functions manage logical record locks:

NWClearLogicalRecord	Unlocks the specified logical record and removes it from the log table.
NWClearLogicalRecordSet	Unlocks and then removes all logical records logged in the log table.
NWLogLogicalRecord	Logs the specified logical record in the log table.
NWLockLogicalRecordSet	Locks all logical records logged in the log table.
NWReleaseLogicalRecord	Unlocks the specified logical record but doesn't remove it from the log table.
NWReleaseLogicalRecordSet	Unlocks all log logical records but doesn't remove them from the log table.

20.2 Semaphores

Like a logical record lock, a semaphore is used to control access to data stored on the NetWare® server. As with logical records, you are responsible for defining and enforcing any restrictions associated with a semaphore. A common use for semaphores is to limit the number of users who can access a network application.

Unlike a logical record, a semaphore allows you to configure the number of applications that can access the data. You can scan for the semaphores that a connection has open, as well as scan for the connections that have opened a semaphore.

To set up a semaphore, call NWOpenSemaphore (page 371). It takes the name of the semaphore and an initial semaphore value. The initial value is zero based and indicates the number of applications that can access the semaphore. For example, if the initial value is 4, five applications can access the semaphore (one of which is the application that opened the semaphore).

After the semaphore is open, applications needing access to the resource associated with the semaphore must call NWWaitOnSemaphore (page 401), which decrements the semaphore. If the resulting value is zero or greater, the function returns successfully. In that case, the resource is considered available. If the semaphore reaches a negative value, the application must wait until the semaphore returns to zero before accessing the resource.

When an application finishes using the protected resource, it calls NWCloseSemaphore (page 352), which decrements the semaphore's open count by one. The semaphore is deleted by the last process to call this function.

20.2.1 Semaphore Functions

These functions manage semaphores:

NWCloseSemaphore	Closes a semaphore and decrements the open count.
NWExamineSemaphore	Returns the semaphore value and the number of workstations that have the semaphore open.
NWOpenSemaphore	Creates and initializes a named semaphore to the specified value.
NWSignalSemaphore	Increments the semaphore value by one.
NWWaitOnSemaphore	Allows the application to queue up for access to the resource associated with a semaphore.

20.3 Synchronization Scan Functions

These functions scan for synchronization information in association with workstation connections:

NWScanLogicalLocksByConn	Scans for all logical record locks on a specified connection.
NWScanLogicalLocksByName	Scans for all record locks on a specified logical name.
NWScanPhysicalLocksByConnFile	Scans for all physical record locks on a specified connection for a specified file.
NWScanPhysicalLocksByFile	Scans for all record locks on a specific physical file.

NWScanSemaphoresByConn	Scans information about the semaphores that a specified connection has open.
NWScanSemaphoresByName	Scans information about a semaphore by name.

Synchronization Tasks

21

This documentation describes common tasks associated with Synchronization.

21.1 Logging Files

Locking procedures are built around the file log table. The NetWare® server maintains a log table for each connection task. (In multi-tasking environments, one task's log table is not affected by another's.)

Call NWLogFileLock2 (page 362) to log files in the log table. Call this function for each file you intend to lock. The `timeOutLimit` parameter lets you control the amount of time the server spends attempting to lock the file. You can also have the server enter the file into the table without attempting to lock it.

21.2 Clearing Logged Files

Files remain in the file log table until you clear them.

- NWClearFileLock2 (page 340) clears a single file from the log table.
- NWClearFileLockSet (page 342) clears all files from the log table.

If a file is locked when you ask the server to clear it from the table, the server releases the lock and clears the file. The server also closes the file if it's open.

21.3 Locking Data and Files

Locking procedures are built around the file log table. The NetWare® server maintains a log table for each connection task. (In multi-tasking environments, one task's log table is not affected by another's.)

To lock one or more files, log the files into the table and then request a file lock. If the server can't lock all the files, the operation fails and none of the files are locked. This method protects your application from entering deadlock with another application, a situation in which each application is waiting for the other to release a partially locked set of files.

The following steps explain file locking, but the basic steps are the same for locking files, physical records, or logical records.

1 Call NWLogFileLock2 (page 362) for each file you intend to lock. This will log each file into the log file. The `timeOut` parameter controls the duration of the server's efforts.

2 After you have logged all files that you intend to lock, call NWLockFileLockSet (page 356). This will lock all the files at once. Again, a timeout value controls the duration of the server's efforts.

3 Use the files you have locked.

4 Release the locks on the files you have locked. To release individual locks, call NWReleaseFileLock2 (page 373) for each file. To release the entire set of files, call NWReleaseFileLockSet (page 375).

5 Clear the files from the file log table. To clear individual files, call NWClearFileLock2 (page 340) for each file. To clear the entire set of files, call NWClearFileLockSet (page 342).

NOTE: If a file is locked when you ask the server to clear it from the log table, the server releases the lock and clears the file. The server also closes the file if it's open.

21.4 Locking Files

After you have logged all the files you intend to lock, call NWLockFileLockSet (page 356). This function locks all the files at once. The value of the `timeOut` parameter controls the duration of the server's efforts.

To lock one or more files, log the files into the table and then request a file lock. If the server can't lock all the files, the operation fails and none of the files are locked. This method protects your application from entering deadlock with another application, a situation in which each application is waiting for the other to release a partially locked set of files.

21.5 Releasing Locked Files

Files remain locked until you specifically ask the server to release them. You can release a file lock on an individual file or on an entire set of locked files:

- NWReleaseFileLock2 (page 373) releases a lock on a single file.
- NWReleaseFileLockSet (page 375) releases the lock on a set of files.

Synchronization Functions

22

This documentation alphabetically lists the synchronization functions and describes their purpose, syntax, parameters, and return values.

NWClearFileLock2

Unlocks the specified file and removes it from the log table

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwfile.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY(NWCCODE) NWClearFileLock2 (
   NWCONN_HANDLE        conn,
   NWDIR_HANDLE         dirHandle,
   const nstr8 N_FAR  *path);
```

Delphi Syntax

```
uses calwin32

Function NWClearFileLock2
  (conn : NWCONN_HANDLE;
   dirHandle : NWDIR_HANDLE;
   const path : pnstr8
) : NWCCODE;
```

Parameters

conn

   (IN) Specifies the NetWare server connection handle.

dirHandle

   (IN) Specifies the directory handle of the directory containing the locked file.

path

   (IN) Points to the string containing the name and path of the locked file.

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x890A	NLM_INVALID_CONNECTION
0x8996	SERVER_OUT_OF_MEMORY
0x8998	VOLUME_DOES_NOT_EXIST
0x899B	BAD_DIRECTORY_HANDLE
0x899C	INVALID_PATH
0x89A1	DIRECTORY_IO_ERROR
0x89FD	BAD_STATION_NUMBER
0x89FF	LOCK_ERROR

Remarks

To avoid deadlock, a workstation must request those resources it needs to lock by making an entry in the File Log Table at the NetWare server. Once the log table is complete, the application attempts to lock those records. Locking works only if all records in the table are available. If some of the logged resources cannot be locked, the lock fails and none of the resources are locked.

If the file is open, NWClearFileLock2 causes it to be closed on the server. The application should close the associated file on the workstation to clear the local file handle correctly.

`path` can specify either a file's complete path name or a path relative to the current working directory. For example, if a file's complete path name is SYS:ACCOUNT/DOMEST/TARGET.DAT and the directory handle mapping is SYS:ACCOUNT, `path` could point to either of the following:

```
SYS:ACCOUNT/DOMEST/TARGET.DAT
DOMEST/TARGET.DAT
```

NCP Calls

0x2222 07 Clear File

See Also

NWClearFileLockSet

Unlocks all files logged in the File Log Table and removes them from the log table

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwfile.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY( NWCCODE )NWClearFileLockSet  (
   void);
```

Delphi Syntax

```
uses calwin32

Function NWClearFileLockSet
  : NWCCODE;
```

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x890A	NLM_INVALID_CONNECTION

Remarks

To avoid deadlock, a workstation must request those resources it needs to lock by making an entry in the File Log Table at the NetWare server. Once the log table is complete, the application attempts to lock those records. Locking works only if all records in the table are available. If some of the logged resources cannot be locked, the lock fails and none of the resources are locked.

All open files in the task's log table are closed. The file handles on the workstation itself are not cleared—this should be done by the application and any error codes should be ignored. NWClearFileLockSet is ignored if the associated task on the workstation does not have logged files.

NCP Calls

0x2222 23 17 Get File Server Information
0x2222 23 22 Get Station's Logged Info (old)
0x2222 23 28 Get Station's Logged Info
0x2222 104 1 Ping for NDS NCP

See Also

NWClearFileLock2 (page 340), NWLogFileLock2 (page 362), NWReleaseFileLock2 (page 373), NWReleaseFileLockSet (page 375)

NWClearLogicalRecord

Unlocks a logical record and removes it from the log table

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwfile.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY(NWCCODE) NWClearLogicalRecord (
   NWCONN_HANDLE       conn,
   const nstr8 N_FAR  *logRecName);
```

Delphi Syntax

```
uses calwin32

Function NWClearLogicalRecord
  (conn : NWCONN_HANDLE;
   const logRecName : pnstr8
) : NWCCODE;
```

Parameters

conn

   (IN) Specifies the NetWare server connection handle containing the logical record.

logRecName

   (IN) Points to the name of the logical record being cleared (128 characters).

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION

0x890A	NLM_INVALID_CONNECTION
0x89FF	LOCK_ERROR

Remarks

A logical record is simply a name (a string) registered with the NetWare server. The name (as with a semaphore) can then be locked or unlocked by applications and can be used as an inter-application locking mechanism.

NOTE: Locking or unlocking a logical record does not physically lock or unlock those resources associated with the logical record; only the applications using the record know about such an association.

Applications define logical record names. A logical record name represents a group of files, physical records, structures, etc. NWLogLogicalRecord or NWLockLogicalRecordSet lock one or more logical record names, not the actual files, physical records, or structures associated with each logical record name. Any uncooperative application can ignore a lock on the logical record name and directly lock physical files or records. Therefore, applications using logical record locks must not use other locking techniques simultaneously.

NCP Calls

0x2222 11 Clear Logical Record

See Also

NWClearLogicalRecordSet (page 346), NWLockLogicalRecordSet (page 358), NWLogLogicalRecord (page 365), NWReleaseLogicalRecord (page 377), NWReleaseLogicalRecordSet (page 379)

NWClearLogicalRecordSet

Unlocks and then removes all of the logical records logged in the log table

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwfile.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY( NWCCODE )NWClearLogicalRecordSet  (
   void);
```

Delphi Syntax

```
uses calwin32

Function NWClearLogicalRecordSet
  : NWCCODE;
```

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL

Remarks

A logical record is simply a name (a string) registered with the NetWare server. The name (as with a semaphore) can then be locked or unlocked by applications and can be used as an inter-application locking mechanism.

NOTE: Locking or unlocking a logical record does not physically lock or unlock those resources associated with the logical record; only the applications using the record know about such an association.

If the requesting process does not have logged logical records, NWClearLogicalRecordSet is ignored.

NCP Calls

0x2222 23 17 Get File Server Information
0x2222 23 22 Get Station's Logged Info (old)
0x2222 23 28 Get Station's Logged Info
0x2222 104 1 Ping for NDS NCP

See Also

NWClearLogicalRecord (page 344), NWLockLogicalRecordSet (page 358), NWLogLogicalRecord (page 365), NWReleaseLogicalRecord (page 377), NWReleaseLogicalRecordSet (page 379)

NWClearPhysicalRecord

Unlocks the specified physical record and removes it from the log table

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwfile.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY( NWCCODE )NWClearPhysicalRecord  (
   NWFILE_HANDLE   fileHandle,
   nuint32         recStartOffset,
   nuint32         recSize);
```

Delphi Syntax

```
uses calwin32

Function NWClearPhysicalRecord
  (fileHandle : NWFILE_HANDLE;
   recStartOffset : nuint32;
   recSize : nuint32
) : NWCCODE;
```

Parameters

fileHandle

> (IN) Specifies the file handle associated with the file containing the physical record being cleared.

recStartOffset

> (IN) Specifies the offset, from the beginning of the file, at which the record starts.

recSize

> (IN) Specifies the length, in bytes, of the locked record.

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL
0x8988	INVALID_FILE_HANDLE
0x89FF	LOCK_ERROR

Remarks

NWClearPhysicalRecord locates the physical record within the specified file by passing the offset in `recStartOffset` and the length in `recSize`.

NOTE: Locking or unlocking a logical record does not physically lock or unlock those resources associated with the logical record; only the applications using the record know about such an association.

`recStartOffset` and `recSize` should match the corresponding parameters in NWLogPhysicalRecord.

NWClearPhysicalRecord is ignored if the requesting workstation does not have logged physical records.

NCP Calls

0x2222 30 Sync Clear Physical Record

See Also

NWClearPhysicalRecordSet (page 350), NWLockPhysicalRecordSet (page 360), NWLogPhysicalRecord (page 368), NWReleasePhysicalRecord (page 381), NWReleasePhysicalRecordSet (page 383)

NWClearPhysicalRecordSet

Unlocks and removes all physical records from the log table

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwfile.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY( NWCCODE )NWClearPhysicalRecordSet  (
   void);
```

Delphi Syntax

```
uses calwin32

Function NWClearPhysicalRecordSet
  : NWCCODE;
```

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL

Remarks

NWClearPhysicalRecordSet is ignored if the requesting workstation does not have logged or locked physical records.

NCP Calls

0x2222 23 17 Get File Server Information
0x2222 23 22 Get Station's Logged Info (old)
0x2222 23 28 Get Station's Logged Info
0x2222 104 1 Ping for NDS NCP

See Also

NWClearPhysicalRecord (page 348), NWLockPhysicalRecordSet (page 360), NWLogPhysicalRecord (page 368), NWReleasePhysicalRecord (page 381), NWReleasePhysicalRecordSet (page 383)

NWCloseSemaphore

Closes a semaphore and decrements the open count of the semaphore, indicating one less process is holding the semaphore open

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwsync.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY( NWCCODE )NWCloseSemaphore  (
   NWCONN_HANDLE    conn,
   nuint32          semHandle);
```

Delphi Syntax

```
uses calwin32

Function NWCloseSemaphore
  (conn : NWCONN_HANDLE;
   semHandle : nuint32
) : NWCCODE;
```

Parameters

conn

> (IN) Specifies the NetWare® server connection handle.

semHandle

> (IN) Specifies the semaphore handle obtained when the semaphore was opened by NWOpenSemaphore.

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL

0x8801	INVALID_CONNECTION
0x890A	NLM_INVALID_CONNECTION
0x89FF	INVALID_SEMAPHORE_HANDLE, LOCK_ERROR

Remarks

If the requesting process is the last process to have this semaphore open, the semaphore is deleted.

NCP Calls

0x2222 32 4 Close Semaphore

See Also

NWExamineSemaphore (page 354), NWOpenSemaphore (page 371), NWSignalSemaphore (page 399), NWWaitOnSemaphore (page 401)

NWExamineSemaphore

Returns the semaphore value

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwsync.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY( NWCCODE )NWExamineSemaphore  (
   NWCONN_HANDLE   conn,
   nuint32         semHandle,
   pnint16         semValue,
   pnuint16        semOpenCount);
```

Delphi Syntax

```
uses calwin32

Function NWExamineSemaphore
  (conn : NWCONN_HANDLE;
   semHandle : nuint32;
   semValue : pnint16;
   semOpenCount : pnuint16
) : NWCCODE;
```

Parameters

conn

   (IN) Specifies the NetWare server connection handle.

semHandle

   (IN) Specifies the semaphore handle obtained when the semaphore was opened by NWOpenSemaphore.

semValue

   (OUT) Points to the current semaphore value (optional).

semOpenCount

(OUT) Points to the number of stations that currently have this semaphore open.

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x890A	NLM_INVALID_CONNECTION
0x89FF	LOCK_ERROR

Remarks

A semaphore value greater than 0 indicates the application can access the associated network resource. A negative value indicates the number of processes waiting to use the semaphore. If the semaphore value is negative, the application must either enter a waiting queue by calling NWWaitOnSemaphore or temporarily abandon its attempt to access the network resource.

`semOpenCount` indicates the number of processes holding the semaphore open. NWOpenSemaphore increments this value. NWCloseSemaphore decrements this value.

`semValue` is optional. Use NULL if a return value is not desired.

NCP Calls

0x2222 32 1 Examine Semaphore

See Also

NWCloseSemaphore (page 352), NWOpenSemaphore (page 371), NWSignalSemaphore (page 399), NWWaitOnSemaphore (page 401)

NWLockFileLockSet

Locks files that have been logged by a workstation task in the File Log Table of a NetWare server

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwfile.h>
or
#include <nwcalls.h>


N_EXTERN_LIBRARY( NWCCODE )NWLockFileLockSet  (
   nuint16   timeOut);
```

Delphi Syntax

```
uses calwin32

Function NWLockFileLockSet
  (timeOut : nuint16
) : NWCCODE;
```

Parameters

timeOut

   (IN) Specifies the length of time the NetWare server attempts to lock the record set before timing out.

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL
0x89FE	TIMEOUT_FAILURE

Remarks

To help avoid deadlock, a workstation task can log file locks in the File Log Table of a NetWare server. When the files in the log table are needed, NWLockFileLockSet can be called.

NWLockFileLockSet will attempt to lock the logged set on all attached servers. Locks will be attempted by ordering the servers according to their net node addresses and making the request on each server. If the request fails at any point, NWLockFileLockSet will automatically release all locks made to that point.

All files on all servers must be available for NWLockFileLockSet to complete successfully.

There is no way to determine which server the lock request failed on.

`timeOut` is the length of time the NetWare server will attempt the operation before failing. This limit is specified in units of 1/18 second (0 = no wait).

NCP Calls

0x2222 23 17 Get File Server Information
0x2222 23 22 Get Station's Logged Info (old)
0x2222 23 28 Get Station's Logged Info
0x2222 104 1 Ping for NDS NCP

See Also

NWClearFileLock2 (page 340), NWClearFileLockSet (page 342), NWLogFileLock2 (page 362), NWReleaseFileLock2 (page 373), NWReleaseFileLockSet (page 375)

NWLockLogicalRecordSet

Locks all logical records logged in the log table

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwfile.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY( NWCCODE )NWLockLogicalRecordSet  (
   nuint8    lockFlags,
   nuint16   timeOut);
```

Delphi Syntax

```
uses calwin32

Function NWLockLogicalRecordSet
  (lockFlags : nuint8;
   timeOut : nuint16
) : NWCCODE;
```

Parameters

lockFlags

   (IN) Specifies the lock flags.

timeOut

   (IN) Specifies the length of time the NetWare server attempts to lock the record set before timing out.

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL
0x89FE	TIMEOUT_FAILURE

0x89FF	LOCK_ERROR

Remarks

Applications define logical record names. A logical record name represents a group of files, physical records, or data structures. NWLogLogicalRecord and NWLockLogicalRecordSet affect one or more logical record names, not the actual files, physical records, or data structures associated with each logical record name. Any uncooperative application can ignore a lock on the logical record name and directly lock physical files or records. Therefore, applications using logical record locks must not simultaneously use other locking techniques.

To avoid deadlock, request the resources needed to lock by making an entry in the File Log Table at the NetWare server. Once the log table is complete, the application attempts to lock those records. The locking works only if all records in the table are available. If some of the logged resources cannot be locked, the lock fails and none of the resources are locked.

A logical record is simply a name (a string) registered with the NetWare server. The name (as with a semaphore) can then be locked or unlocked by applications and can be used as an inter-application locking mechanism.

NOTE: Locking or unlocking a logical record does not physically lock or unlock those resources associated with the logical record; only the applications using the record know about such an association.

`lockFlags` is interpreted as follows:

```
0x00   Lock record with a shareable lock
0x01   Lock record with an exclusive loc
```

`timeOut` is specified in units of 1/18 second (0 = no wait).

NCP Calls

0x2222 23 17 Get File Server Information
0x2222 23 22 Get Station's Logged Info (old)
0x2222 23 28 Get Station's Logged Info
0x2222 104 1 Ping for NDS NCP

See Also

NWClearLogicalRecord (page 344), NWClearLogicalRecordSet (page 346), NWLogLogicalRecord (page 365), NWReleaseLogicalRecord (page 377), NWReleaseLogicalRecordSet (page 379)

NWLockPhysicalRecordSet

Locks all records logged in the physical record log table

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwfile.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY( NWCCODE )NWLockPhysicalRecordSet  (
   nuint8    lockFlags,
   nuint16   timeOut);
```

Delphi Syntax

```
uses calwin32

Function NWLockPhysicalRecordSet
  (lockFlags : nuint8;
   timeOut : nuint16
) : NWCCODE;
```

Parameters

lockFlags

   (IN) Specifies the lock flags.

timeOut

   (IN) Specifies the length of time the NetWare server attempts to lock the record set before
   timing out.

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL
0x89FE	TIMEOUT_FAILURE

Remarks

A physical record lock, as opposed to a logical lock, is the actual lock of a specified record relative to a physical file. Before a record is locked, it is also entered in the File Log Table at the NetWare server. Records can be locked only if all records in the log table are available for locking. This avoids deadlock.

To avoid deadlock, request those resources needing to be locked by making an entry in the File Log Table at the NetWare server. Once the log table is complete, NWLockPhysicalRecordSet attempts to lock those records. The locking only works if all records in the table are available. If some of the logged resources cannot be locked, the lock fails and none of the resources are locked.

`timeOut` is specified in units of 1/18 second (0 = no wait).

`lockFlags` is interpreted as follows:

```
0x00   Lock records with exclusive lock
0x02   Lock records with shareable lock
```

A shareable lock prevents any process, including the one which made the lock, from writing to the record.

NWLockPhysicalRecordSet cannot lock a record that is already locked exclusively by another application. If one or more records, identified in the log table, are already exclusively locked by another application, the attempt to lock the set fails.

NCP Calls

0x2222 23 17 Get File Server Information
0x2222 23 22 Get Station's Logged Info (old)
0x2222 23 28 Get Station's Logged Info
0x2222 104 1 Ping for NDS NCP

See Also

NWClearPhysicalRecord (page 348), NWClearPhysicalRecordSet (page 350), NWLogPhysicalRecord (page 368), NWReleasePhysicalRecord (page 381), NWReleasePhysicalRecordSet (page 383)

NWLogFileLock2

Logs the specified file in the File Log Table and locks the file if the lock flag is set

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwfile.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY(NWCCODE) NWLogFileLock2 (
   NWCONN_HANDLE        conn,
   NWDIR_HANDLE         dirHandle,
   const nstr8 N_FAR   *path,
   nuint8               lockFlags,
   nuint16              timeOut);
```

Delphi Syntax

```
uses calwin32

Function NWLogFileLock2
  (conn : NWCONN_HANDLE;
   dirHandle : NWDIR_HANDLE;
   const path : pnstr8;
   lockFlags : nuint8;
   timeOut : nuint16
) : NWCCODE;
```

Parameters

conn

   (IN) Specifies the NetWare server connection handle.

dirHandle

   (IN) Specifies the directory handle in which the file to be logged resides.

path

   (IN) Points to the string containing the path and file name of the file to be logged.

lockFlags

(IN) Specifies the lock flags.

timeOut

(IN) Specifies the length of time the NetWare server attempts to log the specified file before timing out.

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x890A	NLM_INVALID_CONNECTION
0x897F	ERR_LOCK_WAITING
0x8982	NO_OPEN_PRIVILEGES
0x8996	SERVER_OUT_OF_MEMORY
0x8998	VOLUME_DOES_NOT_EXIST
0x899B	BAD_DIRECTORY_HANDLE
0x899C	INVALID_PATH
0x89FE	TIMEOUT_FAILURE
0x89FF	LOCK_ERROR

Remarks

NWLogFileLock2 logs the specified file for exclusive use by the workstation. If bit 0 of `lockFlags` is set, the server immediately attempts to lock the file.

`lockFlags` ' values are interpreted as follows:

```
0x00   Log file
0x01   Log and lock the file
```

When `lockFlags` is 1, the server attempts to lock the file for the length of time specified by `timeOutLimit`.

`path` can specify either a file's complete path name or a path relative to the current working directory. For example, if a file's complete path name is SYS:ACCOUNT/DOMEST/TARGET.DAT and the directory handle mapping is SYS:ACCOUNT, `path` could point to either of the following:

```
SYS:ACCOUNT/DOMEST/TARGET.DAT
DOMEST/TARGET.DAT
```

`timeOut` is specified in units of 1/18 second (0 = no wait).

NWLogFileLock2 cannot lock files already logged and exclusively locked by other applications. A file can be locked by a client even if the file does not yet exist. This reserves the file name for use by the client locking it.

The File Log Table contains data locking information used by a NetWare server. The NetWare server tracks this information for each workstation and process. Whenever a file, logical record, or physical record is logged, information identifying the data being logged is entered in the log table. Normally, a set of files or records is logged and then locked as a set. However, a single file or record can also be locked when it is entered in the table.

When using log tables, a task first logs all of the files or records that are needed to complete a transaction. The task then attempts to lock the logged set of files or records. If some of the logged resources cannot be locked, the lock fails and none of the resources are locked.

NCP Calls

0x2222 03 Log File
0x2222 23 17 Get File Server Information

See Also

NWClearFileLock2 (page 340), NWClearFileLockSet (page 342), NWLockFileLockSet (page 356), NWReleaseFileLock2 (page 373)

NWLogLogicalRecord

Logs a logical record in a log table

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwfile.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY(NWCCODE)NWLogLogicalRecord (
   NWCONN_HANDLE        conn,
   const nstr8 N_FAR  *logRecName,
   nuint8               lockFlags,
   nuint16              timeOut);
```

Delphi Syntax

```
uses calwin32

Function NWLogLogicalRecord
  (conn : NWCONN_HANDLE;
   const logRecName : pnstr8;
   lockFlags : nuint8;
   timeOut : nuint16
) : NWCCODE;
```

Parameters

conn

   (IN) Specifies the NetWare server connection handle.

logRecName

   (IN) Points to the name of the logical record being logged (128 characters).

lockFlags

   (IN) Specifies the lock flags.

timeOut

(IN) Specifies the length of time the NetWare server attempts to lock the record before timing out.

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x890A	NLM_INVALID_CONNECTION
0x8996	SERVER_OUT_OF_MEMORY
0x89FE	TIMEOUT_FAILURE
0x89FF	LOCK_ERROR

Remarks

A logical record is simply a name (string) registered with the NetWare server. The name (as with a semaphore) can then be locked or unlocked by applications and can be used as an inter-application locking mechanism.

NOTE: Locking or unlocking a logical record does not physically lock or unlock those resources associated with the logical record; only the applications using the record know about such an association.

When `lockFlags` is set to option one or three, the NetWare server attempts to lock the logical record for the length of time specified by `timeOut`. `timeOut` is specified in 1/18 second units.

`lockFlags` ' values are the following:

```
0 = Log only
1 = Log and lock exclusive
3 = Log and lock shareable
```

`timeOut` is specified in units of 1/18 second (0 = no wait).

Normally, a set of files or records is logged and then locked as a set. However, a single file or record can also be locked when it is placed in the log table. The release functions, NWReleaseLogicalRecord and NWReleaseLogicalRecordSet, are used to unlock a lock (or set of locks). The clear functions, NWClearLogicalRecord and NWClearLogicalRecordSet, are used to unlock and remove a lock (or set of locks) from the log table.

To avoid deadlock, request those resources needing to be locked by making an entry in the File Log Table at the NetWare server. Once the log table is complete, NWLogLogicalRecord attempts to lock those records. Locking works only if all records in the table are available. If some of the logged resources cannot be locked, the lock fails and none of the resources are locked.

NWLogLogicalRecord cannot lock files already logged and exclusively locked by other applications.

NCP Calls

0x2222 09 Log Logical Record

See Also

NWClearLogicalRecord (page 344), NWClearLogicalRecordSet (page 346), NWLockLogicalRecordSet (page 358), NWReleaseLogicalRecord (page 377), NWReleaseLogicalRecordSet (page 379)

NWLogPhysicalRecord

Logs a physical record in preparation for a lock

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwfile.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY( NWCCODE )NWLogPhysicalRecord  (
   NWFILE_HANDLE   fileHandle,
   nuint32         recStartOffset,
   nuint32         recLength,
   nuint8          lockFlags,
   nuint16         timeOut);
```

Delphi Syntax

```
uses calwin32

Function NWLogPhysicalRecord
  (fileHandle : NWFILE_HANDLE;
   recStartOffset : nuint32;
   recLength : nuint32;
   lockFlags : nuint8;
   timeOut : nuint16
) : NWCCODE;
```

Parameters

fileHandle

   (IN) Specifies the file handle of the file whose record is being logged (must be valid).

recStartOffset

   (IN) Specifies the offset into the file where the record being logged begins.

recLength

   (IN) Specifies the length, in bytes, of the record to be logged.

lockFlags

(IN) Specifies the lock flags.

timeOut

(IN) Specifies the length of time the NetWare server attempts to lock the record before timing out.

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL
0x0006	INVALID_HANDLE
0x8988	INVALID_FILE_HANDLE
0x8996	SERVER_OUT_OF_MEMORY
0x89FD	LOCK_COLLISION
0x89FE	TIMEOUT_FAILURE
0x89FF	LOCK_ERROR

Remarks

The NetWare server attempts to log the record for the length of time specified by `timeOutLimit` before returning a time out error. `timeOut` is specified in units of 1/18 second (0 = no wait).

`lockFlags`' values follow:

```
0 = Log only
1 = Log and lock exclusive
3 = Log and lock shareable
```

`timeOut` is specified in units of 1/18 second (0 = no wait).

Normally, a set of files or records is logged and then locked as a set. However, a single file or record can also be locked when it is entered in the log table.

The release functions, NWReleasePhysicalRecord and NWReleasePhysicalRecordSet, unlock a lock or set of locks. The clear functions, NWClearPhysicalRecord and NWClearPhysicalRecordSet, unlock and remove a lock or set of locks from the log table.

To avoid deadlock, request those resources needing to be locked by making an entry in the File Log Table at the NetWare server. Once the log table is complete, NWLogPhysicalRecord can then lock those records. The locking works only if all records in the table are available. If some of the logged resources cannot be locked, the lock fails and none of the resources are locked.

NWLogPhysicalRecord returns 0x0006 if an invalid file handle is passed to the `fileHandle` parameter.

NCP Calls

0x2222 26 Log Physical Record

See Also

NWOpenSemaphore

Creates and initializes a named semaphore to the indicated value

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwsync.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY(NWCCODE)NWOpenSemaphore (
   NWCONN_HANDLE        conn,
   const nstr8 N_FAR  *semName,
   nint16               initSemHandle,
   pnuint32             semHandle,
   pnuint16             semOpenCount);
```

Delphi Syntax

```
uses calwin32

Function NWOpenSemaphore
  (conn : NWCONN_HANDLE;
   const semName : pnstr8;
   initSemHandle : nint16;
   semHandle : pnuint32;
   semOpenCount : pnuint16
) : NWCCODE;
```

Parameters

conn

   (IN) Specifies the NetWare server connection handle.

semName

   (IN) Points to the name of the semaphore to be opened.

initSemHandle

   (IN) Specifies the number of tasks that can simultaneously access the resources to which the semaphore is tied.

semHandle

>   (OUT) Points to the NetWare semaphore handle.

semOpenCount

>   (OUT) Points to the number of stations that currently have this semaphore open (optional; set to NULL if you do not wish this number to be returned).

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x890A	NLM_INVALID_CONNECTION
0x8996	SERVER_OUT_OF_MEMORY
0x89FF	LOCK_ERROR

Remarks

Calling NWOpenSemaphore increments the `semOpenCount` counter. If the semaphore exists, `initSemHandle` is ignored. The handle returned must be used to access the semaphore. Only the first application to open the semaphore (and thus create the semaphore) can set the initial value in `initSemHandle`.

NWOpenSemaphore is usually called by setting `initSemHandle` to a value other than 0. If `initSemHandle` is set to 0, consider the following items:

- Semaphore ownership will not be established until the semaphore is signaled.
- The semaphore cannot be used until it is first signaled by an application.
- Usually semaphore applications loop from waiting on a semaphore to signaling the semaphore. If `initSemHandle` is 0, the semaphore must be signaled from outside the wait/signal loop.

NWWaitOnSemaphore decrements the semaphore value by 1 if it is greater than 0. If the semaphore value and the `timeOutValue` parameter are both 0, a time out failure (LOCK_ERROR) will be returned.

NWSignalSemaphore increments the semaphore value by 1.

NCP Calls

0x2222 32 Open Semaphore

See Also

NWCloseSemaphore (page 352), NWExamineSemaphore (page 354), NWSignalSemaphore (page 399), NWWaitOnSemaphore (page 401)

NWReleaseFileLock2

Unlocks the specified file but does not remove it from the log table

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwfile.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY(NWCCODE) NWReleaseFileLock2 (
   NWCONN_HANDLE        conn,
   NWDIR_HANDLE         dirHandle,
   const nstr8 N_FAR  *path);
```

Delphi Syntax

```
uses calwin32

Function NWReleaseFileLock2
  (conn : NWCONN_HANDLE;
   dirHandle : NWDIR_HANDLE;
   path : pnstr8
) : NWCCODE;
```

Parameters

conn

   (IN) Specifies the NetWare server connection handle.

dirHandle

   (IN) Specifies the directory handle of the new directory's root directory.

path

   (IN) Points to the string containing the name and path of the new directory.

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x890A	NLM_INVALID_CONNECTION
0x8998	VOLUME_DOES_NOT_EXIST
0x899B	BAD_DIRECTORY_HANDLE
0x899C	INVALID_PATH

Remarks

`path` can specify either a file's complete path name or a path relative to the current working directory. For example, if a file's complete path name is SYS:ACCOUNT/DOMEST/TARGET.DAT and the directory handle mapping is SYS:ACCOUNT, `path` could be either of the following:

```
SYS:ACCOUNT/DOMEST/TARGET.DAT or
DOMEST/TARGET.DAT
```

NWReleaseFileLock2 is ignored if the requesting workstation does not have locked files.

NCP Calls

0x2222 05 Release File

See Also

NWClearFileLock2 (page 340), NWClearFileLockSet (page 342), NWLogFileLock2 (page 362), NWReleaseFileLockSet (page 375)

NWReleaseFileLockSet

Unlocks all files logged in the log table but does not remove them from the table

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwfile.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY( NWCCODE )NWReleaseFileLockSet  (
   void);
```

Delphi Syntax

```
uses calwin32

Function NWReleaseFileLockSet
  : NWCCODE;
```

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL

Remarks

To avoid deadlock, a workstation must request those files it needs to lock; it does so by making an entry into the File Log Table at the NetWare server. Once the log table is complete, the application can then lock those files. The locking works only if all files in the table are available.

NWReleaseFileLockSet is ignored if the requesting workstation does not have locked files.

NCP Calls

0x2222 23 17 Get File Server Information
0x2222 23 22 Get Station's Logged Info (old)

0x2222 23 28 Get Station's Logged Info

0x2222 104 1 Ping for NDS NCP

See Also

NWClearFileLock2 (page 340), NWClearFileLockSet (page 342), NWLogFileLock2 (page 362), NWReleaseFileLock2 (page 373)

NWReleaseLogicalRecord

Unlocks a logical record but does not remove it from the log table

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwfile.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY(NWCCODE) NWReleaseLogicalRecord (
   NWCONN_HANDLE        conn,
   const nstr8 N_FAR  *logRecName);
```

Delphi Syntax

```
uses calwin32

Function NWReleaseLogicalRecord
  (conn : NWCONN_HANDLE;
   logRecName : pnstr8
) : NWCCODE;
```

Parameters

conn

   (IN) Specifies the NetWare server connection handle containing the logical record.

logRecName

   (IN) Points to the name of the logical record being released (128 characters).

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION

0x890A	NLM_INVALID_CONNECTION
0x89FF	LOCK_ERROR

Remarks

A logical record is simply a name (a string) registered with the NetWare server. The name (as with a semaphore) can then be locked or unlocked by applications and can be used as an inter-application locking mechanism.

NOTE: Locking or unlocking a logical record does not physically lock or unlock those resources associated with the logical record; only the applications using the record know about such an association.

File Log Table contains data locking information used by a NetWare server. The NetWare server tracks this information for each workstation and workstation task. Whenever a file, logical record, or physical record is logged, information identifying the data being logged is placed in the File Log Table. Normally, a set of files or records is logged and then locked as a set. However, a single file or record can also be locked when it is placed in the table.

NWReleaseLogicalRecord is ignored if the requesting workstation has no records to release.

NCP Calls

 0x2222 12 Release Logical Record

See Also

NWClearLogicalRecord (page 344), NWClearLogicalRecordSet (page 346), NWLockLogicalRecordSet (page 358), NWReleaseLogicalRecordSet (page 379)

NWReleaseLogicalRecordSet

Unlocks all the logical records but does not remove them from the log table

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwfile.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY( NWCCODE )NWReleaseLogicalRecordSet  (
   void);
```

Delphi Syntax

```
uses calwin32

Function NWReleaseLogicalRecordSet
  : NWCCODE;
```

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL

Remarks

A logical record is simply a name (a string) registered with the NetWare server. The name (as with a semaphore) can then be locked or unlocked by applications and can be used as an inter-application locking mechanism.

NOTE: Locking or unlocking a logical record does not physically lock or unlock those resources associated with the logical record; only the applications using the record know about such an association.

To avoid deadlock, a workstation is required to request those files it needs to lock; it does so by making an entry into the File Log Table at the NetWare server. Once the log table is complete, the application can then lock those files. The locking works only if all files in the table are available.

NWReleaseLogicalRecordSet is ignored if the requesting workstation or process does not have locked logical records.

NCP Calls

0x2222 23 17 Get File Server Information
0x2222 23 22 Get Station's Logged Info (old)
0x2222 23 28 Get Station's Logged Info
0x2222 104 1 Ping for NDS NCP

See Also

NWClearLogicalRecord (page 344), NWClearLogicalRecordSet (page 346), NWLockLogicalRecordSet (page 358), NWLogLogicalRecord (page 365), NWReleaseLogicalRecord (page 377)

NWReleasePhysicalRecord

Unlocks the specified physical record currently locked in the log table of the requesting workstation but does not remove it from the table

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwfile.h>
or
#include <nwcalls.h>


N_EXTERN_LIBRARY( NWCCODE )NWReleasePhysicalRecord  (
   NWFILE_HANDLE    fileHandle,
   nuint32          recStartOffset,
   nuint32          recSize);
```

Delphi Syntax

```
uses calwin32

Function NWReleasePhysicalRecord
  (fileHandle : NWFILE_HANDLE;
   recStartOffset : nuint32;
   recSize : nuint32
) : NWCCODE;
```

Parameters

fileHandle

   (IN) Specifies the file handle associated with the file containing the specified record.

recStartOffset

   (IN) Specifies the offset, within the file, where the physical record begins.

recSize

   (IN) Specifies the length, in bytes, of the record being released.

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL
0x8988	INVALID_FILE_HANDLE
0x89FF	LOCK_ERROR

Remarks

A physical record lock, as opposed to a logical lock, is the actual lock of a specified record relative to a physical file. When a record is locked, it is also entered into a log table. Records are allowed to be locked only if all records in the log table are available for locking. This is done to avoid deadlock.

NWReleasePhysicalRecord is ignored if the requesting workstation or process does not have locked physical records.

NCP Calls

0x2222 28 Release Physical Record

See Also

NWClearPhysicalRecord (page 348), NWClearPhysicalRecordSet (page 350), NWLockPhysicalRecordSet (page 360), NWLogPhysicalRecord (page 368), NWReleasePhysicalRecordSet (page 383)

NWReleasePhysicalRecordSet

Unlocks, but does not remove, all records currently logged as physical records in the requesting workstation's log table

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwfile.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY( NWCCODE )NWReleasePhysicalRecordSet  (
   void);
```

Delphi Syntax

```
uses calwin32

Function NWReleasePhysicalRecordSet
  : NWCCODE;
```

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL

Remarks

A physical record lock, as opposed to a logical lock, is the actual lock of a specified record relative to a physical file. When a record is locked, it is also entered into a log table. Records are locked only if all records in the log table are available for locking. This is done to avoid deadlock.

NWReleasePhysicalRecordSet is ignored if the workstation does not have locked physical records.

NCP Calls

0x2222 23 17 Get File Server Information
0x2222 23 22 Get Station's Logged Info (old)

0x2222 23 28 Get Station's Logged Info

0x2222 104 1 Ping for NDS NCP

See Also

NWClearPhysicalRecord (page 348), NWClearPhysicalRecordSet (page 350), NWLockPhysicalRecordSet (page 360), NWLogPhysicalRecord (page 368), NWReleasePhysicalRecord (page 381)

NWScanLogicalLocksByConn

Scans for all logical record locks in a specified connection

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwsync.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY( NWCCODE )NWScanLogicalLocksByConn  (
   NWCONN_HANDLE                 conn,
   NWCONN_NUM                    connNum,
   pnint16                       iterHandle,
   CONN_LOGICAL_LOCK  N_FAR    *logicalLock,
   CONN_LOGICAL_LOCKS  N_FAR   *logicalLocks);
```

Delphi Syntax

```
uses calwin32

Function NWScanLogicalLocksByConn
   (conn : NWCONN_HANDLE;
    connNum : NWCONN_NUM;
    iterHandle : pnint16;
    Var logicalLock : CONN_LOGICAL_LOCK;
    Var logicalLocks : CONN_LOGICAL_LOCKS
) : NWCCODE;
```

Parameters

conn

   (IN) Specifies the NetWare server connection handle.

connNum

   (IN) Specifies the connection number of the logged-in object to be scanned.

iterHandle

   (IN/OUT) Points to the number of the next record to be scanned.

logicalLock

(OUT) Points to CONN_LOGICAL_LOCK (optional).

logicalLocks

(OUT) Points to CONN_LOGICAL_LOCKS.

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x890A	NLM_INVALID_CONNECTION
0x8996	SERVER_OUT_OF_MEMORY
0x89C6	NO_CONSOLE_PRIVILEGES
0x89FD	BAD_STATION_NUMBER
0x88FF	Scan Completed

Remarks

The client must have console operator rights to call NWScanLogicalLocksByConn.

iterHandle should be set to 0 initially. Each subsequent call returns the number of the next record to be scanned. iterHandle returns -1 upon completion and should not be changed during the scan.

CONN_LOGICAL_LOCKS is a buffer and should be passed to subsequent NWScanLogicalLocksByConn calls without modification.

If you pass a non-NULL pointer to logicalLock, CONN_LOGICAL_LOCKS passes one record at a time to CONN_LOGICAL_LOCK. If you pass a NULL pointer to logicalLock, CONN_LOGICAL_LOCKS is filled but no records are passed to CONN_LOGICAL_LOCK.

0x88FF is returned when the last record has been passed to CONN_LOGICAL_LOCK and NWScanLogicalLocksByConn is called subsequently.

NCP Calls

0x2222 23 17 Get File Server Information
0x2222 23 239 Get Logical Records By Connection

NWScanLogicalLocksByName

Scans for all record locks in a specified logical name

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwsync.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY(NWCCODE) NWScanLogicalLocksByName (
   NWCONN_HANDLE         conn,
   const nstr8 N_FAR    *logicalName,
   pnint16               iterHandle,
   LOGICAL_LOCK N_FAR   *logicalLock,
   LOGICAL_LOCKS N_FAR  *logicalLocks);
```

Delphi Syntax

```
uses calwin32

Function NWScanLogicalLocksByName
  (conn : NWCONN_HANDLE;
   logicalName : pnstr8;
   iterHandle : pnint16;
   Var logicalLock : LOGICAL_LOCK;
   Var logicalLocks : LOGICAL_LOCKS
) : NWCCODE;
```

Parameters

conn

   (IN) Specifies the NetWare server connection handle.

logicalName

   (IN) Points to the logical lock name to be scanned.

iterHandle

   (IN/OUT) Points to the number of the next record to be scanned.

logicalLock

(OUT) Points to LOGICAL_LOCK (optional).

logicalLocks

(OUT) Points to LOGICAL_LOCKS.

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x890A	NLM_INVALID_CONNECTION
0x88FF	Scan Completed
0x8996	SERVER_OUT_OF_MEMORY
0x89C6	NO_CONSOLE_PRIVILEGES

Remarks

`iterHandle` should be set to 0 initially. Each subsequent call returns the number of the next record to be scanned. `iterHandle` returns -1 upon completion and should not be changed during the scan.

If `logicalLock` is a NULL pointer, `logicalLocks` returns the records in groups, instead of one by one.

NCP Calls

0x2222 23 17 Get File Server Information
0x2222 23 240 Get Logical Record Information

NWScanPhysicalLocksByConnFile

Scans for all physical record locks by a specified connection on a specified file

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwsync.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY(NWCCODE) NWScanPhysicalLocksByConnFile (
   NWCONN_HANDLE             conn,
   NWCONN_NUM                connNum,
   NWDIR_HANDLE              dirHandle,
   const nstr8 N_FAR        *path,
   nuint8                    dataStream,
   pnint16                   iterHandle,
   CONN_PHYSICAL_LOCK N_FAR  *lock,
   CONN_PHYSICAL_LOCKS N_FAR  *locks);
```

Delphi Syntax

```
uses calwin32

Function NWScanPhysicalLocksByConnFile
  (conn : NWCONN_HANDLE;
   connNum : NWCONN_NUM;
   dirHandle : NWDIR_HANDLE;
   path : pnstr8;
   dataStream : nuint8;
   iterHandle : pnint16;
   Var lock : CONN_PHYSICAL_LOCK;
   Var locks : CONN_PHYSICAL_LOCKS
) : NWCCODE;
```

Parameters

conn

   (IN) Specifies the NetWare server connection handle.

connNum

(IN) Specifies the connection number of the logged-in object to be scanned.

dirHandle

(IN) Specifies the directory handle associated with the desired directory path.

path

(IN) Points to a full file path (or a path relative to `dirHandle`) specifying the file to be checked. The last item must be a file name.

dataStream

(IN) Specifies the Macintosh name space (for 3.11 and above only) or set to 0:

0 Resource Fork
1 Data Fork

iterHandle

(IN/OUT) Points to the number of the next record to be scanned (set to 0 initially).

lock

(OUT) Points to the CONN_PHYSICAL_LOCK structure.

locks

(OUT) Points to the CONN_PHYSICAL_LOCKS structure.

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x890A	NLM_INVALID_CONNECTION
0x88FF	Scan Completed
0x8996	SERVER_OUT_OF_MEMORY
0x89FD	BAD_STATION_NUMBER
0x89FF	FILE_NAME_ERROR, NO_FILES_FOUND_ERROR

Remarks

For 3.x, a client must have console operator rights to call NWScanPhysicalLocksByConnFile or NO_CONSOLE_PRIVILEGES will be returned.

For 4.x and above, a client can call NWScanPhysicalLocksByConnFile to return information about its connection without needing console operator privileges. To return information about other connection numbers, you must have console rights. A client with console privileges can pass any valid connection number to NWScanPhysicalLocksByConnFile and receive information about that connection.

`iterHandle` returns -1 upon completion and must not be changed during the scan.

If `lock` is a NULL pointer, `locks` returns the records in groups, instead of one by one.

NCP Calls

0x2222 23 17 Get File Server Information
0x2222 23 237 Get Physical Record Locks By Connection And File
0x2222 23 244 Convert Path To Entry
0x2222 62 Scan First

NWScanPhysicalLocksByFile

Scans for all record locks in a specified physical file

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwsync.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY(NWCCODE) NWScanPhysicalLocksByFile (
   NWCONN_HANDLE          conn,
   NWDIR_HANDLE           dirHandle,
   const nstr8 N_FAR     *path,
   nuint8                 dataStream,
   pnint16                iterHandle,
   PHYSICAL_LOCK N_FAR   *lock,
   PHYSICAL_LOCKS N_FAR  *locks);
```

Delphi Syntax

```
uses calwin32

Function NWScanPhysicalLocksByFile
  (conn : NWCONN_HANDLE;
   dirHandle : NWDIR_HANDLE;
   path : pnstr8;
   dataStream : nuint8;
   iterHandle : pnint16;
   Var lock : PHYSICAL_LOCK;
   Var locks : PHYSICAL_LOCKS
) : NWCCODE;
```

Parameters

conn

> (IN) Specifies the NetWare server connection handle.

dirHandle

> (IN) Specifies the directory handle associated with the desired directory path.

path

(IN) Points to a full file path (or a path relative to `dirHandle`) specifying the file to be checked.

dataStream

(IN) Specifies the Macintosh name space (for 3.11 and above only) or set to 0:

0   Resource Fork
1   Data Fork

iterHandle

(IN/OUT) Points to the next record to be scanned; must be set to 0 initially.

lock

(OUT) Points to PHYSICAL_LOCK (optional).

locks

(OUT) Points to PHYSICAL_LOCKS.

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x890A	NLM_INVALID_CONNECTION
0x88FF	Scan Completed
0x8996	SERVER_OUT_OF_MEMORY
0x8998	VOLUME_DOES_NOT_EXIST
0x899B	BAD_DIRECTORY_HANDLE
0x899C	INVALID_PATH
0x89C6	NO_CONSOLE_PRIVILEGES
0x89FD	BAD_STATION_NUMBER
0x89FF	NO_FILES_FOUND_ERROR

Remarks

The client must have console operator rights to call NWScanPhysicalLocksByFile.

`iterHandle` returns -1 upon completion, and should not be changed during the scan.

If `lock` is a NULL pointer, `locks` returns the records in groups, instead of one by one.

NCP Calls

0x2222 23 17 Get File Server Information
0x2222 23 238 Get Physical Record Locks By File
0x2222 23 244 Convert Path to Entry

NWScanSemaphoresByConn

Scans information about the semaphores opened by a specified connection

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwsync.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY( NWCCODE )NWScanSemaphoresByConn  (
   NWCONN_HANDLE          conn,
   NWCONN_NUM             connNum,
   pnint16                iterHandle,
   CONN_SEMAPHORE  NWPTR    semaphore,
   CONN_SEMAPHORES  NWPTR   semaphores);
```

Delphi Syntax

```
uses calwin32

Function NWScanSemaphoresByConn
   (conn : NWCONN_HANDLE;
    connNum : NWCONN_NUM;
    iterHandle : pnint16;
    Var semaphore : CONN_SEMAPHORE;
    Var semaphores : CONN_SEMAPHORES
) : NWCCODE;
```

Parameters

conn

   (IN) Specifies the NetWare server connection handle.

connNum

   (IN) Specifies the connection number of the logged-in object to be scanned.

iterHandle

   (IN/OUT) Points to the number of the next record to be scanned; should be set to 0 initially.

semaphore

(OUT) Points to CONN_SEMAPHORE (optional).

semaphores

(OUT) Points to CONN_SEMAPHORES.

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x890A	NLM_INVALID_CONNECTION
0x88FF	Scan Completed
0x8996	SERVER_OUT_OF_MEMORY

Remarks

For 3.x, you must have console operator privileges to call NWScanSemaphoresByConn or NO_CONSOLE_PRIVILEGES will be returned.

For 4.x and above, a client can call NWScanSemaphoresByConn to return information about its connection without needing console operator privileges. To return information about other connection numbers, you must have console rights. A client with console privileges can pass any valid connection number to NWScanSemaphoresByConn and receive information about that connection.

`iterHandle` returns -1 upon completion, and should not be changed during the scan.

If `semaphore` is a NULL pointer, `semaphores` returns the records in a group, instead of one by one.

NWScanSemaphoresByConn returns SUCCESSFUL even when `connNum` is invalid. Call NWGetFileServerInformation to return the `maxConns` supported for the specific server. Only use `connNum` in the range of zero- `maxConns`.

NCP Calls

0x2222 23 17 Get File Server Information
0x2222 23 241 Get Connection's Semaphores

See Also

NWCCGetConnRefInfo, NWGetObjectConnectionNumbers (*NDK: Connection, Message, and NCP Extensions*)

NWScanSemaphoresByName

Scans information about a semaphore by name

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwsync.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY(NWCCODE) NWScanSemaphoresByName (
   NWCONN_HANDLE        conn,
   const nstr8 N_FAR    *semName,
   pnint16              iterHandle,
   SEMAPHORE N_FAR      *semaphore,
   SEMAPHORES N_FAR     *semaphores);
```

Delphi Syntax

```
uses calwin32

Function NWScanSemaphoresByName
   (conn : NWCONN_HANDLE;
    semName : pnstr8;
    iterHandle : pnint16;
    Var semaphore : SEMAPHORE;
    Var semaphores : SEMAPHORES
) : NWCCODE;
```

Parameters

conn

   (IN) Specifies the NetWare server connection handle.

semName

   (IN) Points to the semaphore name to be scanned.

iterHandle

   (IN/OUT) Points to the number of the next record to be scanned; should be set to 0 initially.

semaphore

(OUT) Points to SEMAPHORE (optional).

semaphores

(OUT) Points to SEMAPHORES.

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x890A	NLM_INVALID_CONNECTION
0x88FF	Scan Completed
0x8996	SERVER_OUT_OF_MEMORY
0x89C6	NO_CONSOLE_PRIVILEGES

Remarks

The client must have console operator rights to call NWScanSemaphoresByName.

iterHandle returns -1 upon completion, and should not be changed during the scan.

If semaphore is a NULL pointer, semaphores returns the records in groups, instead of one by one .

NCP Calls

0x2222 23 17 Get File Server Information
0x2222 23 242 Get Semaphore Information

NWSignalSemaphore

Increments the semaphore value by one

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwsync.h>
or
 #include <nwcalls.h>

N_EXTERN_LIBRARY( NWCCODE )NWSignalSemaphore  (
   NWCONN_HANDLE   conn,
   nuint32         semHandle);
```

Delphi Syntax

```
uses calwin32

Function NWSignalSemaphore
  (conn : NWCONN_HANDLE;
   semHandle : nuint32
) : NWCCODE;
```

Parameters

conn

  (IN) Specifies the NetWare server connection handle.

semHandle

  (IN) Specifies the semaphore handle of the semaphore to be signaled (obtained by calling NWOpenSemaphore).

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION

0x890A	NLM_INVALID_CONNECTION
0x89F	LOCK_ERROR

Remarks

If another client is waiting on the semaphore, a successful completion code is returned to the waiting client.

An application must call NWSignalSemaphore when it finishes accessing the network resource associated with the semaphore. If processes are waiting to use the semaphore, the first process in the queue is released (signaled).

NCP Calls

0x2222 32 3 Signal Semaphore

See Also

NWWaitOnSemaphore

Waits on a semaphore for a specified time

NetWare Server: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Platform: NLM, Windows NT, Windows 95, Windows 98

Library: Cross-Platform NetWare Calls (CAL*.*)

Service: Synchronization

Syntax

```
#include <nwsync.h>
or
#include <nwcalls.h>

N_EXTERN_LIBRARY( NWCCODE )NWWaitOnSemaphore  (
   NWCONN_HANDLE   conn,
   nuint32         semHandle,
   nuint16         timeOutValue);
```

Delphi Syntax

```
uses calwin32

Function NWWaitOnSemaphore
  Function NWWaitOnSemaphore
(conn : NWCONN_HANDLE;
   semHandle : nuint32;
   timeOutValue : nuint16
) : NWCCODE;
```

Parameters

conn

   (IN) Specifies the NetWare server connection handle.

semHandle

   (IN) Specifies the semaphore handle returned by calling NWOpenSemaphore.

timeOutValue

   (IN) Specifies the length of time the application will wait for the semaphore.

Return Values

These are common return values; see Return Values (*NDK: Connection, Message, and NCP Extensions*) for more information.

0x0000	SUCCESSFUL

0x8801	INVALID_CONNECTION
0x890A	NLM_INVALID_CONNECTION
0x89FE	TIMEOUT_FAILURE
0x89FF	LOCK_ERROR

Remarks

NWWaitOnSemaphore decrements the semaphore value counter by 1 if it is greater than 0. If the semaphore value counter and the `timeOutValue` parameter are both 0, a time out failure (LOCK_ERROR) will be returned. If the value is 0 before the time out expires, Successful is returned, and the application can access the associated resource.

If the value is <0, NWWaitOnSemaphore queues the application for the time interval specified in `timeOutValue`.

`timeOutValue` indicates how long the NetWare server should wait if the semaphore value is negative. `timeOutValue` is specified in units of 1/18 second (0 = no wait). It has no default value.

See Also

NWCloseSemaphore (page 352), NWExamineSemaphore (page 354), NWOpenSemaphore (page 371), NWSignalSemaphore (page 399)

Synchronization Structures

23

This documentation alphabetically lists the Synchronization structures and describes their purpose, syntax, and fields.

CONN_LOGICAL_LOCK

Returns a connection's logical locks

Service: Synchronization

Defined In: nwsync.h

Structure

```
typedef struct
{
   nuint16   taskNumber;
   nuint8    lockStatus;
   nstr8     logicalName[128];
} CONN_LOGICAL_LOCK;
```

Delphi Structure

```
uses calwin32

CONN_LOGICAL_LOCK = packed Record
    taskNumber : nuint16;
    lockStatus : nuint8;
    logicalName : Array[0..127] Of nstr8;
  End;
```

Fields

taskNumber

　　Specifies the task number of the workstation that has the file open.

lockStatus

　　Specifies a bit mask describing how the file is locked:

　　0x01 Locked
　　0x02 Open shareable
　　0x04 Logged
　　0x08 Open Normal
　　0x40 TTS holding
　　0x80 Transaction flag set

logicalName

　　Specifies the name of the logical lock.

CONN_LOGICAL_LOCKS

Returns a connection's logical lock list

Service: Synchronization

Defined In: nwsync.h

Structure

```
typedef struct
{
   nuint16   nextRequest;
   nuint16   numRecords;
   nuint8    records[508];
   nuint16   curOffset;
   nuint16   curRecord;
} CONN_LOGICAL_LOCKS;
```

Delphi Structure

```
uses calwin32

  CONN_LOGICAL_LOCKS = packed Record
    nextRequest : nuint16;
    numRecords : nuint16;
    records : Array[0..507] Of nuint8;
    curOffset : nuint16;
    curRecord : nuint16;
  End;
```

Fields

nextRequest

Is used internally by NWScanLogicalLocksByConn to retrieve the next request.

numRecords

Specifies the number of CONN_LOGICAL_LOCK structures that are contained in `records`.

records

Specifies an array of CONN_LOGICAL_LOCK structures.

curOffset

Is for internal use only.

curRecord

Is for internal use only.

Remarks

You must not modify the values in `nextRequest`, `curOffset` and `curRecord`; they are used internally to return the next record in a request.

CONN_PHYSICAL_LOCK

Returns a connection's physical locks

Service: Synchronization

Defined In: nwsync.h

Structure

```
typedef struct
{
   nuint16   taskNumber;
   nuint8    lockType;
   nuint32   recordStart;
   nuint32   recordEnd;
} CONN_PHYSICAL_LOCK;
```

Delphi Structure

```
uses calwin32

CONN_PHYSICAL_LOCK = packed Record
    taskNumber : nuint16;
    lockType : nuint8;
    recordStart : nuint32;
    recordEnd : nuint32;
  End;
```

Fields

taskNumber

Specifies the number of the task using the file.

lockType

Specifies if the file is locked with the following bits being set:

none Not locked
Bit 0 Locked exclusive
Bit 1 Locked shareable
Bit 2 Logged
Bit 6 Lock held by TTS

recordStart

Specifies the byte offset of where the record begins in the file.

recordEnd

Specifies the byte offset of where the record ends in the file.

CONN_PHYSICAL_LOCKS

Returns a connection's physical lock list

Service: Synchronization

Defined In: nwsync.h

Structure

```
typedef struct
{
   nuint16              nextRequest;
   nuint16              numRecords;
   CONN_PHYSICAL_LOCK   locks[51];
   nuint16              curRecord;
   nuint8               reserved[22];
} CONN_PHYSICAL_LOCKS;
```

Delphi Structure

```
uses calwin32

CONN_PHYSICAL_LOCKS = packed Record
    nextRequest : nuint16;
    numRecords : nuint16;
    locks : Array[0..50] Of CONN_PHYSICAL_LOCK;
    curRecord : nuint16;
    reserved : Array[0..21] Of nuint8;
  End;
```

Fields

nextRequest

> Is used internally by NWScanPhysicalLocksByConnFile to retrieve the next request.

numRecords

> Specifies the number of valid PHYSICAL_LOCK structures contained in `locks`.

locks

> Specifies an array of CONN_PHYSICAL_LOCK structures.

curRecord

> Specifies the current PHYSICAL_LOCK to return in `lock`.

reserved

> Is reserved for future use.

Remarks

You must not modify the values in `nextRequest`, `curOffset` and `curRecord`; they are used internally to return the next record in a request.

CONN_SEMAPHORE

Returns semaphore information list

Service: Synchronization

Defined In: nwsync.h

Structure

```
typedef struct
{
   nuint16   openCount;
   nuint16   semaphoreValue;
   nuint16   taskNumber;
   nstr8     semaphoreName[128];
} CONN_SEMAPHORE;
```

Delphi Structure

```
uses calwin32

 CONN_SEMAPHORE = packed Record
    openCount : nuint16;
    semaphoreValue : nuint16;
    taskNumber : nuint16;
    semaphoreName : Array[0..127] Of nstr8;
  End;
```

Fields

openCount

Specifies the number of connections that have the semaphore open for use.

semaphoreValue

Specifies the current value of the semaphore.

taskNumber

Specifies the number of the connection's task that is using the semaphore.

semaphoreName

Specifies the name of the semaphore.

CONN_SEMAPHORES

Returns a connection's semaphore list

Service: Synchronization

Defined In: nwsync.h

Structure

```
typedef struct
{
   nuint16   nextRequest;
   nuint16   numRecords;
   nuint8    records[508];
   nuint16   curOffset;
   nuint16   curRecord;
} CONN_SEMAPHORES;
```

Delphi Structure

```
uses calwin32

  CONN_SEMAPHORES = packed Record
    nextRequest : nuint16;
    numRecords : nuint16;
    records : Array[0..507] Of nuint8;
    curOffset : nuint16;
    curRecord : nuint16;
  End;
```

Fields

nextRequest

   Is used internally by NWScanSemaphoresByConn to retrieve the next record.

numRecords

   Specifies the number of CONN_SEMAPHORE structures that are contained in `records`.

records

   Specifies a buffer of CONN_SEMAPHORE structures.

curOffset

   Is for internal use only.

curRecord

   Is for internal use only.

Remarks

You must not modify the values in `nextRequest`, `curOffset` and `curRecord`; they are used internally to return the next record in a request.

LOGICAL_LOCK

Defines logical lock information

Service: Synchronization

Defined In: nwsync.h

Structure

```
typedef struct
{
   NWCONN_NUM   connNumber;
   nuint16      taskNumber;
   nuint8       lockStatus;
} LOGICAL_LOCK;
```

Delphi Structure

```
uses calwin32

  LOGICAL_LOCK = packed Record
    connNumber : nuint16;
    taskNumber : nuint16;
    lockStatus : nuint8;
  End;
```

Fields

connNumber

Specifies the logical connection that is using the logical record.

taskNumber

Specifies the task number of the workstation that has the file open.

lockStatus

Specifies a bit mask describing how the file is locked:

0x01 Locked
0x02 Open shareable
0x04 Logged
0x08 Open Normal
0x40 TTS holding
0x80 Transaction flag set

LOGICAL_LOCKS

Returns a list of logical locks

Service: Synchronization

Defined In: nwsync.h

Structure

```
typedef struct
{
   nuint16        useCount;
   nuint16        shareableLockCount;
   nuint8         locked;
   nuint16        nextRequest;
   nuint16        numRecords;
   LOGICAL_LOCK   logicalLock[128];
   nuint16        curRecord;
} LOGICAL_LOCKS;
```

Delphi Structure

```
uses calwin32

 LOGICAL_LOCKS = packed Record
    useCount : nuint16;
    shareableLockCount : nuint16;
    locked : nuint8;
    nextRequest : nuint16;
    numRecords : nuint16;
    logicalLock : Array[0..127] Of LOGICAL_LOCK;
    curRecord : nuint16
  End;
```

Fields

useCount

Specifies the number of logical connections that have logged the logical record.

shareableLockCount

Specifies the number of logical connections that have a shareable lock on the logical record.

locked

Specifies whether the logical record is exclusively locked:

0 Not exclusively locked
1 Exclusively locked

nextRequest

Is used internally by NWScanLogicalLocksByName to retrieve the next request.

numRecords

> Specifies the number of LOGICAL_LOCK structures.

logicalLock

> Specifies a buffer containing the LOGICAL_LOCK structure.

curRecord

> Is for internal use only.

Remarks

You must not modify the values in `nextRequest` and `curRecord`; they are used internally to return the next record in a request.

PHYSICAL_LOCK

Returns physical lock information

Service: Synchronization

Defined In: nwsync.h

Structure

```
typedef struct
{
   nuint16   loggedCount;
   nuint16   shareableLockCount;
   nuint32   recordStart;
   nuint32   recordEnd;
   nuint16   connNumber;
   nuint16   taskNumber;
   nuint8    lockType;
} PHYSICAL_LOCK;
```

Delphi Structure

```
uses calwin32

PHYSICAL_LOCK = packed Record
   {$IfDef N_ARCH_32}
      loggedCount        : nuint16;
   {$else}
      loggedCount        : nuint8;
   {$EndIf}
      shareableLockCount : nuint16;
      recordStart        : nuint32;
      recordEnd          : nuint32;
      connNumber         : nuint16;
      taskNumber         : nuint16;
      lockType           : nuint8;
   {$IfDef N_ARCH_32}
      filler             : nuint8;
      filler2            : nuint16;
   {$EndIf}
     End;
```

Fields

loggedCount

   Specifies the number of tasks having the record logged.

shareableLockCount

   Specifies the number of tasks having the record locked shareable.

recordStart

Specifies the byte offset of where the record begins in the file.

recordEnd

Specifies the logical connection that has exclusively locked the record.

connNumber

Specifies the logical connection number for the connection that has the record exclusively locked.

taskNumber

Specifies the task number for the logical connection that has the record exclusively locked.

lockType

Specifies whether the record is locked:

0x00 Not locked
0xFE Locked by a file lock
0xFF Locked by begin share file set

PHYSICAL_LOCKS

Returns a list of physical locks

Service: Synchronization

Defined In: nwsync.h

Structure

```
typedef struct
{
   nuint16         nextRequest;
   nuint16         numRecords;
   PHYSICAL_LOCK   locks[32];
   nuint16         curRecord;
   nuint8          reserved[8];
} PHYSICAL_LOCKS;
```

Delphi Structure

```
uses calwin32

PHYSICAL_LOCKS = packed Record
   nextRequest : nuint16;
   numRecords : nuint16;
   locks : Array[0..31] Of PHYSICAL_LOCK;
   curRecord : nuint16;
   reserved : Array[0..7] Of nuint8;
End;
```

Fields

nextRequest

   Is used internally by NWScanPhysicalLocksByFile to retrieve the next request.

numRecords

   Specifies the number of valid PHYSICAL_LOCK structures.

locks

   Specifies an array of PHYSICAL_LOCK structures.

curRecord

   Specifies the current PHYSICAL_LOCK structure.

reserved

   Is reserved for future use.

SEMAPHORE

Returns semaphore information

Service: Synchronization

Defined In: nwsync.h

Structure

```
typedef struct
{
   NWCONN_NUM    connNumber;
   nuint16       taskNumber;
} SEMAPHORE;
```

Delphi Structure

```
uses calwin32

  SEMAPHORE = packed Record
    connNumber : nuint16;
    taskNumber : nuint16;
  End;
```

Fields

connNumber

   Specifies the logical connection number of the connection that is using the semaphore.

taskNumber

   Specifies the task number of the logical connection that has the semaphore open.

SEMAPHORES

Returns a list of semaphores

Service: Synchronization

Defined In: nwsync.h

Structure

```
typedef struct
{
   nuint16     nextRequest;
   nuint16     openCount;
   nuint16     semaphoreValue;
   nuint16     semaphoreCount;
   SEMAPHORE   semaphores[170];
   nuint16     curRecord;
} SEMAPHORES;
```

Delphi Structure

```
uses calwin32

SEMAPHORES = Record
   nextRequest : nuint16;
   openCount : nuint16;
   semaphoreValue : nuint16;
   semaphoreCount : nuint16;
   semaphores : Array[0..169] Of SEMAPHORE;
   curRecord : nuint16;
End;
```

Fields

nextRequest

Is used internally by NWScanSemaphoresByName to retrieve the next request.

openCount

Specifies the number of logical connections that have the semaphore open.

semaphoreValue

Specifies the current value of the semaphore (-127 to 128).

semaphoreCount

Specifies the number of SEMAPHORE structures.

semaphores

Specifies a buffer of SEMAPHORE structures.

curRecord

Is for internal use only.

Remarks

You must not modify the values in `nextRequest` and `curRecord`; they are used internally to return the next record in a request.

Server-Based AFP Concepts

24

This documentation describes Server-Based AFP, its functions, and features.

NOTE: Server-Based AFP functions are included for backwards compatibility. It is strongly recommended that new applications use File Engine Services functions instead.

Server-Based AFP functions allow applications to create, access, and delete Mac OS format directories and files on a NetWare® server.

The Mac OS name space driver must be loaded and the Mac OS name space must be added to each desired volume.

24.1 File-Naming Conventions

The following is a discussion of the file-naming conventions that apply to Server-Based AFP in a NetWare environment. Server-Based AFP directory and file names (long names) are 1 to 31 characters long. A long name is a Delphi string preceded by one byte specifying the length of the name. Names can contain any ASCII character from 1 to 255 except a colon or a NULL character. A server automatically generates a DOS-style name (short name) to correspond to each Server-Based AFP directory or filename created. Therefore, all Server-Based AFP directory and file names on a server have both a long name and a short name. To convert a long name with no periods to a short name, the server selects the first eight DOS-valid characters in the long name:

Long Name

THIS IS A NAME

Short Name

THISISAN

If the long name contains a period within the first nine characters, the server selects the first eight DOS-valid characters before the period and the first three after the period, excluding another period:

Long Name

THIS.IS.A.NAME

Short Name

THIS.IS

This renaming mechanism permits both Mac OS and other workstations to access Server-Based AFP directories and files. However, suppose an application creates the following two files in a parent directory:

Long Name

THIS IS THE FIRST FILE

THIS IS THE SECOND FILE

Since their short names are the same (THISISTH and THISISTH), the server adds an ascending decimal number to the short name of the second file: THISIST1. Now, suppose a Mac OS workstation copies THIS IS THE SECOND FILE (THISIST1) to a floppy diskette, and then copies the file from the diskette to a different directory on the server. Since the short name of THIS IS THE SECOND FILE no longer conflicts with another short name, the server changes the short name of the file to THISISTH.

Unlike NetWare directory paths that contain slashes or backslashes to separate directory and file names, Server-Based AFP paths contain NULL values (decimal zeros):

NetWare Path:

volume:directory/directory/file

Server-Based AFP Path:

volume:directory0directory0file

Applications frequently target the short name of a directory or file with a combination of a NetWare directory handle and a short (NetWare style) directory or file path. Likewise, applications target the long name of a directory or file with a combination of an Server-Based AFP entry ID and a long (Server-Based AFP style) directory or file path. Applications should not combine NetWare directory handles and long names, or Server-Based AFP entry IDs and short names. (The text of each function explains more about Server-Based AFP entry IDs.)

24.2 Server-Based AFP Functions

These are the Server-Based AFP functions:

AFPAllocTemporaryDirHandle	Maps a NetWare directory handle to an AFP directory
AFPCreateDirectory	Creates a directory with an AFP directory name
AFPCreateFile	Creates a file with an AFP filename
AFPDelete	Deletes a file or directory
AFPDirectoryEntry	Determines whether a directory or file is in AFP (long) form
AFPGetEntryIDFromName	Returns the AFP entry ID for an AFP file or directory
AFPGetEntryIDFromNetWareHandle	Returns an AFP entry ID for a specified file
AFPGetEntryIDFromPathName	Maps an AFP entry ID to a NetWare directory or file path
AFPGetFileInformation	Returns information about the AFP side of a file or directory
AFPOpenFileFork	Opens an AFP file fork from a DOS environment
AFPRename	Moves and/or renames a file or directory
AFPScanFileInformation	Returns information about an AFP directory or file
AFPSetFileInformation	Sets information for an AFP file or directory
AFPSupported	Determines whether a server supports AFP functions

Server-Based Extended Attribute Functions

25

This documentation alphabetically lists the server-based extended attribute functions and describes their purpose, syntax, parameters, and return values.

For cross-platform functionality, see Developing NLMs with Cross-Platform Functions (*NDK: NLM Development Concepts, Tools, and Functions*) and call the alternative function listed with each NLM function.

CloseEA

Closes a file that was opened for EA I/O by OpenEA

Local Servers: blocking

Remote Servers: blocking

Classification: 3.12, 3.2, 4.x, 5.x, 6.x

Service: Server-Based Extended Attributes

Syntax

```
#include <\nlm\nit\nwextatt.h>

int CloseEA (
   int   handle);
```

Parameters

handle

   (IN) Specifies the EA handle opened by OpenEA.

Return Values

0	Success
Nonzero	Failure

Remarks

For cross-platform functionality, call NWCloseEA (page 154).

Values for EAs are defined in the \nlm\nit\nwextatt.h file.

NOTE: This function does not work remotely on NetWare 3.x servers.

See Also

OpenEA (page 432), ReadEA (page 434), WriteEA (page 436)

CopyEA

Copies EAs from one file or directory to another

Local Servers: blocking

Remote Servers: blocking

Classification: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Service: Server-Based Extended Attributes

Syntax

```
#include <\nlm\nit\nwextatt.h>

int CopyEA (
   const char  *srcPath,
   const char  *destPath,
   int          destVolumeNumber,
   LONG         destDirectoryNumber,
   LONG        *EAccount,
   LONG        *EAdataSize,
   LONG        *EAkeySize);
```

Parameters

srcPath

   (IN) Specifies the source directory/file. Relative paths must be relative to the current working directory (CWD).

destPath

   (IN) Specifies the path of destination directory/file. Relative paths must be relative to the next two parameters.

destVolumeNumber

   (IN) Specifies the volume number of destination (only used if the volume is not specified in the `destPath`)

destDirectoryNumber

   (IN) Specifies the directory number of destination (only used if the directory is not specified in the `destPath`).

EAcount

   (OUT) Receives the number of EAs that were copied.

EAdataSize

   (OUT) Receives the amount of EA data that was copied.

EAkeySize

   (OUT) Receives the amount of EA key data that was copied.

Return Values

If successful, this function returns zero. Otherwise, a nonzero value is returned.

Remarks

This function does not have a cross-platform counterpart.

All of the source EAs are copied to the destination.

SetCurrentNameSpace sets the name space which is used for parsing the path input to this function.

NOTE: This function does not work remotely on NetWare 3.x servers.

See Also

OpenEA (page 432), ReadEA (page 434), WriteEA (page 436)

EnumerateEA

Enumerates extended attributes (EAs) in a directory or file

Local Servers: blocking

Remote Servers: blocking

Classification: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Service: Server-Based Extended Attributes

Syntax

```
#include <\nlm\nit\nwextatt.h>

int EnumerateEA (
   int          handle,
   const char  *keyBuffer,
   char        *dataBuffer,
   LONG         dataBufferSize,
   int          startPosition,
   LONG        *dataSize,
   LONG        *EAsInReply);
```

Parameters

handle

    (IN) Specifies the handle of the file or directory whose EAs are to be enumerated.

keyBuffer

    (IN) Specifies the key of an EA defined in the file or directory or else NULL to retrieve all EA information (maximum size is 255 bytes).

dataBuffer

    (OUT) Receives the requested information. Either a single instance of the T_enumerateEAwithKey (page 441) structure if a key was specified or multiple instances (one for every EA) of T_enumerateEAnoKey (page 440).

dataBufferSize

    (IN) Specifies the size of `dataBuffer`. Only as much information as can fit is returned (there must be room to return information for at least one EA). If there is not enough room for at least one EA, `EAsInReply` is 0.

startPosition

    (IN) If no key is specified, the starting position identifies the EA for which to start returning information. Normally, the starting position is 0.

dataSize

    (OUT) Returns the number of bytes of information returned in `dataBuffer`.

EAsInReply

(OUT) Returns the number of EAs for which information was returned.

Return Values

This function returns a value of 0 if successful. Otherwise, it returns an error code (nonzero value).

Remarks

This function is used to determine which EAs are defined for a particular file or directory or how big an EA or set of EAs is.

If a key value is specified for `keyBuffer`, only information about that key is returned in `dataBuffer`.

If NULL or an empty key is specified, information about all the keys is returned in `dataBuffer`.

This function does not work for remote NetWare 3.x servers.

See Also

GetEAInfo (page 431), ReadEA (page 434), WriteEA (page 436)

GetEAInfo

Returns information about the EAs for a particular open file

Local Servers: blocking

Remote Servers: blocking

Classification: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Service: Server-Based Extended Attributes

Syntax

```
#include <\nlm\nit\nwextatt.h>

int GetEAInfo (
   int    handle,
   LONG  *totalEAs,
   LONG  *totalDataSizeOfEAs,
   LONG  *totalKeySizeOfEAs);
```

Parameters

handle

   (IN) Specifies the EA handle of the (open) file to return EA information about.

totalEAs

   (OUT) Returns the number of EAs defined for the file.

totalDataSizeOfEAs

   (OUT) Returns the sum of the sizes of the data portion of all the EAs.

totalKeySizeOfEAs

   (OUT) Returns the sum of the sizes of the key portion of all the EAs.

Return Values

This function returns a value of 0 if successful. Otherwise, it returns an error code (nonzero value).

Remarks

This function is returned to get basic information about the set of EAs of a particular file. This information can be used along with the function EnumerateEA to get more detailed information about the EAs of a file.

See Also

CloseEA (page 426), EnumerateEA (page 429), OpenEA (page 432)

OpenEA

Opens a file or directory for EA I/O

Local Servers: blocking

Remote Servers: blocking

Classification: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Service: Server-Based Extended Attributes

Syntax

```
#include <\nlm\nit\nwextatt.h>

int OpenEA (
   const char  *path,
   LONG         reserved);
```

Parameters

path

   (IN) Path of file or directory whose extended attributes are to be read, written, or scanned.

reserved

   (IN) Is reserved; set to zero.

Return Values

This function returns an EA handle if successful. Otherwise, it returns:

EFAILURE	Failure `errno` and `NetWareErrno` are set.

Remarks

For cross-platform functionality, call NWOpenEA (page 170).

The EA I/O functions that use an EA handle are CloseEA, EnumerateEA, GetEAInfo, ReadEA, and WriteEA.

SetCurrentNameSpace sets the name space which is used for parsing the path input to this function.

If OpenEA fails, more information is available in `NetWareErrno`.

NOTE: OpenEA does not work on remote NetWare 3.x servers.

NetWare 3.12 and 4.1

If an incorrect server name is passed, 252 (FCh) is returned. If an incorrect volume name is passed, 152 (98h) is returned. For both of these naming errors, `errno` and `NetWareErrno` are both set to zero.

See Also

CloseEA (page 426), ReadEA (page 434), WriteEA (page 436)

ReadEA

Reads EAs

Local Servers: blocking

Remote Servers: blocking

Classification: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Service: Server-Based Extended Attributes

Syntax

```
#include <\nlm\nit\nwextatt.h>

int ReadEA (
    int          handle,
    const char  *keyBuffer,
    char        *dataBuffer,
    LONG         dataBufferSize,
    LONG        *accessFlags);
```

Parameters

handle

   (IN) Specifies the EA handle of the directory or file from which to read an EA.

keyBuffer

   (IN) Specifies the ASCIIZ string. Contains key of EA to read. The maximum length of this buffer is 255 bytes.

dataBuffer

   (OUT) Specifies the buffer into which the data portion of the EA is read.

dataBufferSize

   (IN) Specifies the size of `dataBuffer`. This value must be a multiple of 128 bytes.

accessFlags

   (OUT) Returns access flags (set by WriteEA).

Return Values

This function returns the number of bytes read (1- 64 KB) if successful. Otherwise, it returns:

EFAILURE	Failure (`errno` and `NetWareErrno` are set).

Remarks

For cross-platform functionality, call NWReadEA (page 174).

The directory or file must first be opened for extended attribute I/O with OpenEA. ReadEA reads the EA specified by the key in `keyBuffer`. The EA is read into `dataBuffer`. The return value of the function is the number of bytes that are actually read. The `accessFlags` parameter is the user-defined value set by WriteEA.

The whole EA must be read with one call to ReadEA. If the `dataBuffer` is too small, an error and no data is returned. The maximum length of data is 64 KB. EnumerateEA can be used to determine how long an EA's data value is.

This function does not work on remote NetWare 3.x servers.

See Also

CloseEA (page 426), OpenEA (page 432), WriteEA (page 436)

WriteEA

Writes EAs

Local Servers: blocking

Remote Servers: blocking

Classification: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Service: Server-Based Extended Attributes

Syntax

```
#include <\nlm\nit\nwextatt.h>

int WriteEA (
   int          handle,
   const char  *keyBuffer,
   const char  *dataBuffer,
   LONG         dataBufferSize,
   LONG         accessFlags);
```

Parameters

handle

   (IN) Specifies the EA handle of the directory or file to write an EA for.

keyBuffer

   (IN) Specifies an ASCIIZ string. Contains the key of EA to write. The maximum length of this buffer is 255 bytes.

dataBuffer

   (IN) Specifies the buffer that contains the data to write to the EA.

dataBufferSize

   (IN) Specifies the size of `dataBuffer`. This value must be a multiple of 128 bytes.

accessFlags

   (IN) Specifies the user-defined access flags that can be used for setting additional information.

Return Values

Values for EAs are defined in nwfattr.h

This function returns the number of bytes written (equal to `dataBufferSize`) if successful. Otherwise, it returns:

EFAILURE	Failure (`errno` and `NetWareErrno` are set).

Remarks

For cross-platform functionality, call NWWriteEA (page 180).

The directory or file must first be opened for extended attribute I/O with OpenEA. WriteEA writes the EA specified by the key in `keyBuffer`. The EA's data is assumed to be in `dataBuffer`. The whole EA must be written with one call to WriteEA.

The return value of the function is the number of bytes actually written.

An EA can be deleted by setting `dataBufferSize` to 0 and `dataBuffer` to NULL.

This function does not work on remote NetWare 3.x servers.

See Also

CloseEA (page 426), OpenEA (page 432), ReadEA (page 434)

Server-Based Extended Attribute Structures

26

This documentation alphabetically lists the server-based extended attributes structures and describes their purpose, syntax, and fields.

T_enumerateEAnoKey

Describes the data layout returned by the EnumerateEA function when a key value is not specified

Service: Extended Attribute

Defined In: nwextatt.h

Structure

```
typedef struct
{
   LONG   valueLength ;
   WORD   keyLength ;
   LONG   accessFlags ;
   char   keyValue [1];
} T_enumerateEAnoKey;
```

Fields

valueLength

> Specifies the length of the Extended Attribute corresponding to the key.

keyLength

> Specifies the length of the key value, which starts at the keyValue field.

accessFlags

> Specifies developer-defined access flags.

keyValue

> Specifies the first character of the key value.

Remarks

The key is a developer-defined value used for categorizing Extended Attributes.

T_enumerateEAwithKey

Describes the data layout returned by the EnumerateEA function when a key value is specified

Service: Extended Attribute

Defined In: nwextatt.h

Structure

```
typedef struct
{
   LONG   EALength ;
   WORD   keyLength ;
   LONG   accessFlags ;
   LONG   keyExtants ;
   LONG   valueExtants ;
   char   keyValue [1];
} T_enumerateEAwithKey;
```

Fields

EALength

Specifies the length of the Extended Attribute.

keyLength

Specifies the length of the keyValue field.

accessFlags

Specifies developer-defined access flags.

keyExtants

Specifies the number of 128-byte extants used by the key value.

valueExtants

Specifies the number of 128-byte extants used by the Extended Attribute.

keyValue

Specifies the first character of the key value.

Remarks

The key is a developer-defined value used for categorizing Extended Attributes.

Server-Based Synchronization Concepts

27

This documentation describes Server-Based Synchronization, its functions, and features.

Synchronization functions enable applications to coordinate access to network files and other resources. These services are divided into two categories: Locking and Semaphores.

27.1 Locking

The NetWare® OS provides calls that allow applications to lock files, physical records, or logical records. Before locking a file/record, an application must record the following information about the file/record in a log table residing on the server:

- Name (for files)
- Location and size (for records)

This log table is associated exclusively with the requesting task. Once all files/records are logged, the application makes a call to lock everything identified in the log table. If all logged files/records are available, the server locks them. However, if one or more files/records are in use by another application, the request fails.

Applications can log physical or logical records before locking the records. The files/records may also be locked as they are logged.

The technique of logging file/record sets (as opposed to single items) before locking them ensures that either all necessary files/records in a set are locked, or none of the files/records are locked. This prevents a phenomenon called deadlock or deadly embrace.

Deadlock occurs when two or more applications attempt to lock the same files. For example, if application_1 locks file AAA and attempts to lock file BBB and simultaneously, application_2 locks BBB and attempts to lock AAA, a deadlock occurs. In this situation, both applications wait indefinitely for the other application to release the needed file. Neither application releases the file it has already locked. Thus, the applications wait in a deadly embrace until the server is rebooted.

Applications can specify a time-out value when making a call to lock a set of files or records. The time-out value is counted in units of approximately 1/18 of a second. If one or more files or records are unavailable at the time the application makes the locking call, the time-out count allows the program to wait for a second or two in case a locked file or record becomes available. This reduces retries and, ultimately, network traffic.

Unlike a physical record lock, a logical record lock does not actually lock bytes. Instead, a logical record lock acts somewhat like a semaphore. Applications cooperatively define a logical record name that represents a group of files, records, structures, and so on. When an application locks a logical record, it only locks the logical record name, and not the group of files, records, or structures the name represents. Any uncooperative application can ignore a lock on the logical record name and directly lock the physical files or records. Therefore, applications using logical record locks should not use other locking techniques simultaneously on the same data.

The release functions are used to unlock a record (or set of records). The clear functions are used to unlock and remove a record (or set of records) from the log table.

27.2 Semaphores

NetWare provides calls that enable applications to create, open, examine, and close semaphores. Applications can also use semaphore functions to increment and decrement the value associated with a semaphore. Like a logical record lock, a semaphore is a name associated with network resources such as files, records, structures, and so on.

Both logical record locks and semaphores limit the number of applications that can access network resources at one time:

- "Limiting the Number of Users" on page 444
- "Restricting Access to Resources" on page 445

Logical record locks allow only one application to access a network resource at any one time. Semaphores, however, allow a configurable number of applications (1 to 127) to access a network resource at one time.

The number of semaphores that can be opened corresponds directly to the number of locks that the server can handle for a workstation (connection). The default is 500. If more semaphores are required, the number of locks must be increased using the SET console command.

When an application creates a semaphore, the application assigns a value to the semaphore (for example, 4). The value indicates how many applications can access the resource associated with the semaphore at one time. In the example, four applications can access the resource at one time (1 to 4).

After opening an existing semaphore, an application first checks the value. If the value is greater than or equal to zero, the application may check the value by calling ExamineSemaphore or ExamineLocalSemaphore. If the value is greater than zero, the application can access the associated network resource. The application decrements the value by calling WaitOnSemaphore and then accesses the resource. After accessing the resource, the application increments the semaphore value by calling SignalSemaphore, and then closes the semaphore.

If an application opens a semaphore and discovers that the value is negative, it cannot access the resource immediately. The application can either wait a specified time-out interval until the resource becomes accessible, or the application can retry later.

In the NetWare environment, semaphores can be used to limit the number of users of a particular resource and to restrict access to a particular resource.

27.2.1 Limiting the Number of Users

To use semaphores to limit the number of users of a resource, use the OpenSemaphore and CloseSemaphore functions only. When OpenSemaphore is called, one of the values returned is the number of processes that have the semaphore open. The program can check the value against a predefined user limit and take appropriate action.

In cases where at most n users are allowed to use n resources simultaneously, if more than n users attempt to use the resource, the excess users are not to be put in a queue to wait for the resources to become free. For example, semaphores can be used to limit the number of users of a program (software licensing).

27.2.2 Restricting Access to Resources

Semaphores can be used to restrict access to resources, allowing only serial access. To request access to a resource, a process opens a semaphore associated with the resource and, through that semaphore, sees whether it is permissible to access the resource. If the resource is unavailable, the calling process is placed on a wait queue; it then waits its turn to access the resource. If the calling process is allowed access to the resource, it signals the semaphore when it is done using the resource. If there are processes waiting for the resource (in the semaphores queue), the first process in the queue is allowed first access to the resource. Typically, in a client-server relationship, servers wait and clients signal.

Restricting access to a resource does not necessarily mean that only one process can access a resource at a time. For example, if the resource being restricted is a modem pool with four modems, the program (semaphore) would allow simultaneous access to the modem pool by four users, but deny access to subsequent requesters.

27.3 Server-Based Synchronization Functions

ClearFile	Unlocks and removes a file from the log table
ClearFileSet	Unlocks and removes all files from the log table
ClearLogicalRecord	Unlocks and removes a logical record from the log table
ClearLogicalRecordSet	Unlocks and removes all logical records from the log table
ClearPhysicalRecord	Unlocks and removes a physical record from the log table
ClearPhysicalRecordSet	Unlocks and removes all physical records from the log table
CloseSemaphore	Decrements a semaphores open count
ExamineSemaphore	Returns the current value and open count of a semaphore
LockFileSet	Attempts to lock all files logged in the log table
LockLogicalRecordSet	Attempts to lock all logical records logged in the log table
LockPhysicalRecordSet	Attempts to lock all physical records logged in the log table
LogFile	Places a file into the log table and optionally locks the file
LogLogicalRecord	Places a logical record string into the log table and optionally locks the record
LogPhysicalRecord	Places a physical record into the log table and optionally locks the record
OpenSemaphore	Opens the specified semaphore, or creates it if it does not exist
ReleaseFile	Unlocks a currently locked file but does not remove it from the log table
ReleaseFileSet	Unlocks all currently locked files but does not remove them from the log table
ReleaseLogicalRecord	Unlocks a currently locked logical record but does not remove it from the log table

ReleaseLogicalRecordSet	Unlocks all currently locked logical records but does not remove them from the log table
ReleasePhysicalRecord	Unlocks a currently locked physical record but does not remove it from the log table
ReleasePhysicalRecordSet	Unlocks all currently locked physical records but does not remove them from the log table
SignalSemaphore	Increments the semaphore value of the specified semaphore
WaitOnSemaphore	Decrements a semaphore value

Server-Based Synchronization Functions

<div style="text-align: right; font-size: 3em;">28</div>

This documentation alphabetically lists the Server-Based Synchronization functions and describes their purpose, syntax, parameters, and return values.

- "ClearFile" on page 448
- "ClearFileSet" on page 449
- "ClearLogicalRecord" on page 450
- "ClearLogicalRecordSet" on page 451
- "ClearPhysicalRecord" on page 452
- "ClearPhysicalRecordSet" on page 454
- "CloseSemaphore" on page 455
- "ExamineSemaphore" on page 456
- "LockFileSet" on page 458
- "LockLogicalRecordSet" on page 459
- "LockPhysicalRecordSet" on page 461
- "LogFile" on page 463
- "LogLogicalRecord" on page 465
- "LogPhysicalRecord" on page 467
- "OpenSemaphore" on page 469
- "ReleaseFile" on page 471
- "ReleaseFileSet" on page 472
- "ReleaseLogicalRecord" on page 473
- "ReleaseLogicalRecordSet" on page 474
- "ReleasePhysicalRecord" on page 475
- "ReleasePhysicalRecordSet" on page 477
- "SignalSemaphore" on page 478
- "WaitOnSemaphore" on page 479

For cross-platform functionality, see Developing NLMs with Cross-Platform Functions (*NDK: NLM Development Concepts, Tools, and Functions*), use the CALNLM32.NLM library, and call the alternative function listed with each NLM function.

ClearFile

Unlocks the specified file and removes it from the log table

Local Servers: blocking

Remote Servers: blocking

Classification: 3.x, 4.x, 5.x, 6.x

Service: Server-Based Synchronization

Syntax

```
#include <\nlm\nit\nwsync.h>


int ClearFile (
   char   *fileName);
```

Parameters

fileName

>   (IN) Specifies the string containing the filename with optional full path specification (maximum 255 characters, including the NULL terminator).

Return Values

Decimal	Hex	Constant	Description
0	(0x00)	ESUCCESS	
255	(0xFF)	ERR_DOS_FILE_NOT_FOUND	No logged file found with the specified filename.

Remarks

For cross-platform functionality, call NWClearFileLock2 (page 340).

The `fileName` parameter can specify either a file's complete path name or a path relative to the current working directory (CWD).

SetCurrentNameSpace sets the name space which is used for parsing the path input to this function.

See Also

ClearFileSet (page 449), LockFileSet (page 458), LogFile (page 463), ReleaseFile (page 471), ReleaseFileSet (page 472)

ClearFileSet

Unlocks and removes all files in the log table

Local Servers: blocking

Remote Servers: blocking

Classification: 3.x, 4.x, 5.x, 6.x

Service: Server-Based Synchronization

Syntax

```
#include <\nlm\nit\nwsync.h>

void ClearFileSet (void);
```

Remarks

For cross-platform functionality, call NWClearFileLockSet (page 342).

Any open files that are in the NLM log table are closed. This function is ignored if there are no logged files.

See Also

ClearFile (page 448), LockFileSet (page 458), LogFile (page 463), ReleaseFile (page 471), ReleaseFileSet (page 472)

ClearLogicalRecord

Unlocks a logical record and removes it from the log table

Local Servers: blocking

Remote Servers: blocking

Classification: 3.x, 4.x, 5.x, 6.x

Service: Server-Based Synchronization

Syntax

```
#include <\nlm\nit\nwsync.h>

int ClearLogicalRecord (
   char   *logicalRecordName);
```

Parameters

logicalRecordName

> (IN) Specifies the string containing the record name (maximum 100 characters, including the NULL terminator).

Return Values

Decimal	Hex	Constant
0	(0x00)	ESUCCESS
255	(0xFF)	ERR_NO_RECORD_FOUND

Remarks

For cross-platform functionality, call ClearLogicalRecord (page 450).

The log table resides on the server and is associated exclusively with the requesting task.

Applications define logical record names. A logical record name represents a group of files, physical records, structures, and so on. When LogLogicalRecord or LockLogicalRecordSet locks one or more logical record names, it does not lock the actual files, physical records, structures, and so on associated with each logical record name. It just locks the logical record name. Any uncooperative application can ignore a lock on the logical record name and directly lock or access physical files or records. Therefore, applications using logical record locks must not use other locking techniques simultaneously.

See Also

ClearLogicalRecordSet (page 451), LockLogicalRecordSet (page 459), LogLogicalRecord (page 465), ReleaseLogicalRecord (page 473), ReleaseLogicalRecordSet (page 474)

ClearLogicalRecordSet

Unlocks all logical records and removes them from the log table

Local Servers: blocking

Remote Servers: blocking

Classification: 3.x, 4.x, 5.x, 6.x

Service: Server-Based Synchronization

Syntax

```
#include <\nlm\nit\nwsync.h>

void ClearLogicalRecordSet (void);
```

Remarks

For cross-platform functionality, call NWClearLogicalRecordSet (page 346).

If there are no logged logical records, this function is ignored.

See Also

ClearLogicalRecord (page 450), LockLogicalRecordSet (page 459), LogLogicalRecord (page 465), ReleaseLogicalRecord (page 473), ReleaseLogicalRecordSet (page 474)

ClearPhysicalRecord

Unlocks a physical record and removes it from the log table

Local Servers: blocking

Remote Servers: blocking

Classification: 3.x, 4.x, 5.x, 6.x

Service: Server-Based Synchronization

Syntax

```
#include <\nlm\nit\nwsync.h>

int ClearPhysicalRecord (
    int    fileHandle,
    long   recordStartOffset,
    long   recordLength);
```

Parameters

fileHandle

   (IN) Specifies the handle of the file containing the record to be cleared.

recordStartOffset

   (IN) Specifies the offset within the file where the record begins.

recordLength

   (IN) Specifies the length of record in bytes.

Return Values

Decimal	Hex	Constant
0	(0x00)	ESUCCESS
255	(0xFF)	ERR_NO_RECORD_FOUND

Remarks

For cross-platform functionality, call NWClearPhysicalRecord (page 348).

The log table resides on the server and is associated exclusively with the requesting task.

The `fileHandle` value is returned by a previous call to open, sopen, creat, or fileno.

The application locates the beginning of the physical record within the specified file by passing an offset in `recordStartOffset`. The application specifies the length of a physical record by passing a length value in `recordLength`.

See Also

ClearPhysicalRecordSet (page 454), LockPhysicalRecordSet (page 461), LogPhysicalRecord (page 467), ReleasePhysicalRecord (page 475), ReleasePhysicalRecordSet (page 477)

ClearPhysicalRecordSet

Unlocks all physical records and removes them from the log table

Local Servers: blocking

Remote Servers: blocking

Classification: 3.x, 4.x, 5.x, 6.x

Service: Server-Based Synchronization

Syntax

```
#include <\nlm\nit\nwsync.h>

void ClearPhysicalRecordSet (void);
```

Remarks

For cross-platform functionality, call ClearPhysicalRecordSet (page 454).

This function is ignored if there are no logged physical records.

See Also

ClearPhysicalRecord (page 452), LockPhysicalRecordSet (page 461), LogPhysicalRecord (page 467), ReleasePhysicalRecord (page 475), ReleasePhysicalRecordSet (page 477)

CloseSemaphore

Decrements a semaphore's open count

Local Servers: blocking

Remote Servers: blocking

Classification: 3.x, 4.x, 5.x, 6.x

Service: Server-Based Synchronization

Syntax

```
#include <\nlm\nit\nwsync.h>

int CloseSemaphore (
   long   semaphoreHandle);
```

Parameters

semaphoreHandle

   (IN) Specifies the semaphore handle of an open semaphore.

Return Values

Decimal	Hex	Constant
0	(0x00)	ESUCCESS
255	(0xFF)	ERR_INVALID_SEMAPHORE_HANDLE

Remarks

For cross-platform functionality, call NWCloseSemaphore (page 352).

This function decrements the open count of the semaphore, indicating that one less process is holding the semaphore open. If the requesting process is the last process to have this semaphore open, the semaphore is deleted. An application can obtain a semaphore handle by calling OpenSemaphore.

See Also

ExamineSemaphore (page 456), OpenSemaphore (page 469), SignalSemaphore (page 478), WaitOnSemaphore (page 479)

ExamineSemaphore

Returns the current value and open count of a local semaphore

Local Servers: nonblocking

Remote Servers: blocking

Classification: 3.x, 4.x, 5.x, 6.x

Service: Server-Based Synchronization

Syntax

```
#include <\nlm\nit\nwsync.h>

int ExamineSemaphore (
   long    semaphoreHandle,
   int    *semaphoreValue,
   WORD   *openCount);
```

Parameters

semaphoreHandle

   (IN) Specifies the semaphore handle of an open semaphore.

semaphoreValue

   (OUT) Receives current semaphore value (-127 to 127).

openCount

   (OUT) Receives the number of processes that currently have the semaphore opened.

Return Values

Decimal	Hex	Constant
0	(0x00)	ESUCCESS
255	(0xFF)	ERR_INVALID_SEMAPHORE_HANDLE

Remarks

For cross-platform functionality, call NWExamineSemaphore (page 354).

The `semaphoreValue` is decremented for each WaitOnSemaphore and incremented for each SignalSemaphore. A positive `semaphoreValue` indicates that the application can access the associated network resource. If `semaphoreValue` is zero or negative, the application must either enter a waiting queue by calling the function WaitOnSemaphore, or temporarily abandon its attempt to access the network resource.

The `openCount` indicates the number of processes holding the semaphore open. OpenSemaphore increments this value. CloseSemaphore decrements this value.

See Also

LockFileSet

Attempts to lock all files in the log table

Local Servers: blocking

Remote Servers: blocking

Classification: 3.x, 4.x, 5.x, 6.x

Service: Server-Based Synchronization

Syntax

```
#include <\nlm\nit\nwsync.h>

int LockFileSet (
   WORD   timeoutLimit);
```

Parameters

timeoutLimit

   (IN) Specifies the timeout value.

Return Values

Decimal	Hex	Constant	Description
0	(0x00)	ESUCCESS	
254	(0xFE)	ERR_TIMEOUT_FAILURE	Timeout limit was reached before all files were available for locking.)
255	(0xFF)	ERR_FAILURE	

Remarks

For cross-platform functionality, call NWLockFileLockSet (page 356).

The timeoutLimit parameter indicates how long the server should wait if it cannot lock all files immediately. The timeoutLimit is specified in units of 1/18 of a second (0 means no wait). This function cannot lock a file that is already exclusively locked by another application. Therefore, if one or more files identified in the log table are already exclusively locked, the attempt fails.

See Also

ClearFile (page 448), ClearFileSet (page 449), LogFile (page 463), ReleaseFile (page 471), ReleaseFileSet (page 472)

LockLogicalRecordSet

Attempts to lock all the logical records in the log table

Local Servers: blocking

Remote Servers: blocking

Classification: 3.x, 4.x, 5.x, 6.x

Service: Server-Based Synchronization

Syntax

```
#include <\nlm\nit\nwsync.h>

int LockLogicalRecordSet (
   WORD   timeoutLimit);
```

Parameters

timeoutLimit

   (IN) Specifies the timeout value.

Return Values

Decimal	Hex	Constant
0	(0x00)	ESUCCESS
254	(0xFE)	ERR_TIMEOUT_FAILURE
		Timeout limit was reached before all records were available for locking.
255	(0xFF)	ERR_FAILURE

Remarks

For cross-platform functionality, call NWLockLogicalRecordSet (page 358).

Applications define logical record names. A logical record name represents a group of files, physical records, structures, and so on. When LogLogicalRecord or LockLogicalRecordSet locks one or more logical record names, it does not lock the actual files, physical records, structures, and so on associated with each logical record name. It just locks the logical record name. Any uncooperative application can ignore a lock on or can access the logical record name and directly lock or access physical files or records. Therefore, applications using logical record locks must not use other locking techniques simultaneously.

The timeoutLimit parameter indicates how long the server should wait if it cannot lock all the records immediately. The timeoutLimit is specified in units of 1/18 of a second (0 means no wait).

The function cannot lock a logical record that is already exclusively locked by another application. Therefore, if one or more logical records identified in the log table are already exclusively locked by another application, the attempt fails.

See Also

ClearLogicalRecord (page 450), ClearLogicalRecordSet (page 451), LogLogicalRecord (page 465), ReleaseLogicalRecord (page 473), ReleaseLogicalRecordSet (page 474)

LockPhysicalRecordSet

Attempts to lock all physical records in the log table

Local Servers: blocking

Remote Servers: blocking

Classification: 3.x, 4.x, 5.x, 6.x

Service: Server-Based Synchronization

Syntax

```
#include <\nlm\nit\nwsync.h>

int LockPhysicalRecordSet (
   BYTE   lockDirective,
   WORD   timeoutLimit);
```

Parameters

lockDirective

   (IN) 0 = Lock records with exclusive locks 1 = Lock records with sharable read-only locks

timeoutLimit

   (IN) Specifies the timeout value.

Return Values

Decimal	Hex	Constant
0	(0x00)	ESUCCESS
254	(0xFE)	ERR_TIMEOUT_FAILURE
		Timeout limit was reached before all records were available for locking.
255	(0xFF)	ERR_FAILURE

Remarks

For cross-platform functionality, call NWLockPhysicalRecordSet (page 360).

The `timeoutLimit` parameter indicates how long the server should wait if it cannot lock all the records immediately. The `timeoutLimit` is specified in units of 1/18 of a second (0 means no wait).

The function cannot lock a record that is already exclusively locked by another application. If one or more records identified in the log table are already exclusively locked by another application, the attempt fails.

See Also

ClearPhysicalRecord (page 452), ClearPhysicalRecordSet (page 454), LogPhysicalRecord (page 467), ReleasePhysicalRecord (page 475), ReleasePhysicalRecordSet (page 477)

LogFile

Logs a file into the log table and optionally locks the file

Local Servers: blocking

Remote Servers: blocking

Classification: 3.x, 4.x, 5.x, 6.x

Service: Server-Based Synchronization

Syntax

```
#include <\nlm\nit\nwsync.h>

int LogFile (
   char    *fileName,
   BYTE    lockDirective,
   WORD    timeoutLimit);
```

Parameters

fileName

   (IN) Specifies the string containing filename with optional full path specification of the file to be logged (maximum 255 characters, including the NULL terminator).

lockDirective

   (IN) Indicates if and how the file should be locked: 0x00 = Log file 0x01 = Log and lock file 0x03 = Log and lock sharable

timeoutLimit

   (IN) Specifies the timeout value (valid only if `lockDirective` is not 0).

Return Values

Decimal	Hex	Constant
0	(0x00)	ESUCCESS
150	(0x96)	SERVER_OUT_OF_MEMORY
254	(0xFE)	ERR_TIMEOUT_FAILURE
		Timeout limit was reached before file was available for locking.
255	(0xFF)	ERR_FAILURE

Remarks

For cross-platform functionality, call .

A log table contains data-locking information used by a server. The server tracks this information. Whenever a file, logical record, or physical record is logged, information identifying the data being logged is placed in the log table. Normally a set of files or records are logged and then locked as a set. However, a single file or record can also be locked when it is placed in the log table.

When using log tables, an application first logs all files or records to complete a transaction. The application then attempts to lock the logged set of files or records. If some of the logged resources cannot be locked, the lock fails and none of the resources are locked. Therefore, either all of the resources needed to complete a transaction are locked or none of the resources are locked. This function cannot lock files that are exclusively locked by other applications.

When the `lockDirective` parameter specifies that the file should be locked when it is logged, the `timeoutLimit` parameter indicates how long the server should wait if it cannot lock the file immediately. The `timeoutLimit` parameter is specified in units of 1/18 of a second (0 means no wait).

The release functions are used to unlock a file (or set of files). The clear functions are used to unlock and remove a file (or set of files) from the log table.

SetCurrentNameSpace sets the name space which is used for parsing the path input to this function.

See Also

ClearFile (page 448), ClearFileSet (page 449), LockFileSet (page 458), ReleaseFile (page 471)

LogLogicalRecord

Logs a logical record into the log table and optionally locks the record

Local Servers: blocking

Remote Servers: blocking

Classification: 3.x, 4.x, 5.x, 6.x

Service: Server-Based Synchronization

Syntax

```
#include <\nlm\nit\nwsync.h>

int LogLogicalRecord (
   char   *logicalRecordName,
   BYTE    lockDirective,
   WORD    timeoutLimit);
```

Parameters

logicalRecordName

(IN) NULL-terminated string containing the record name. (_MAX_LOGREC_NAME, defined in NWSYNC.H, is 128 characters, including the NULL terminator).

lockDirective

(IN) Indicates if and how the record should be locked:

0x00 = Log record.
0x01 = Log and lock record with an exclusive lock.
0x03 = Log and lock record with a sharable read-only lock.

timeoutLimit

(IN) Timeout value (valid only if `lockDirective` is not 0).

Return Values

Decimal	Hex	Constant
0	(0x00)	ESUCCESS
150	(0x96)	ERR_SERVER_OUT_OF_MEMORY
254	(0xFE)	ERR_TIMEOUT_FAILURE
		Timeout limit was reached before record was available for locking.
255	(0xFF)	ERR_FAILURE

Remarks

For cross-platform functionality, call NWLogLogicalRecord (page 365).

A log table contains data-locking information used by a server. The server tracks this information. Whenever a logical record is logged, information identifying the data being logged is placed in the log table. Normally, a set of files or records are logged and then locked as a set. However, a record can also be locked when it is placed in the log table.

When using log tables, an application first logs all records to complete a transaction. The application then attempts to lock the logged set of records. If some of the logged resources cannot be locked, the lock fails and none of the resources are locked. Therefore, either all of the resources needed to complete a transaction are locked or none of the resources are locked. This function cannot lock logical records that are exclusively locked by other applications.

Applications define logical record names. A logical record name represents a group of files, physical records, structures, and so on. When LogLogicalRecord or LockLogicalRecordSet locks one or more logical record names, it does not lock the actual files, physical records, structures, and so on, associated with each logical record name. It just locks the logical record name. Any uncooperative application can ignore a lock on the logical record name and directly lock or access physical files or records. Therefore, applications using logical record locks must not use other locking techniques simultaneously.

When the `lockDirective` parameter specifies that the record should be locked when it is logged, the `timeoutLimit` parameter indicates how long the server should wait if it cannot lock the record immediately. The `timeoutLimit` parameter is specified in units of 1/18 of a second (0 means no wait).

See Also

ClearLogicalRecord (page 450), ClearLogicalRecordSet (page 451), LockLogicalRecordSet (page 459), ReleaseLogicalRecord (page 473), ReleaseLogicalRecordSet (page 474)

LogPhysicalRecord

Logs a physical record into the log table and optionally locks the record

Local Servers: blocking

Remote Servers: blocking

Classification: 3.x, 4.x, 5.x, 6.x

Service: Server-Based Synchronization

Syntax

```
#include <\nlm\nit\nwsync.h>

int LogPhysicalRecord (
   int    fileHandle,
   long   recordStartOffset,
   long   recordLength,
   BYTE   lockDirective,
   WORD   timeoutLimit);
```

Parameters

fileHandle

   (IN) Specifies the handle of the file containing the record to be logged.

recordStartOffset

   (IN) Specifies the offset within the file where the record begins.

recordLength

   (IN) Specifies the length of the record in bytes.

lockDirective

   (IN) Indicates if and how the record should be locked:

   0x00 = Log record.
   0x01 = Log and lock record with an exclusive lock.
   0x03 = Log and lock record with a sharable read-only lock.

timeoutLimit

   (IN) Specifies the timeout value (valid only if `lockDirective` is not 0).

Return Values

Decimal	Hex	Constant
0	(0x00)	ESUCCESS
150	(0x96)	ERR_SERVER_OUT_OF_MEMORY

Decimal	Hex	Constant
254	(0xFE)	ERR_TIMEOUT_FAILURE
		Timeout limit was reached before record was available for locking.
255	(0xFF)	ERR_FAILURE

Remarks

For cross-platform functionality, call NWLogPhysicalRecord (page 368).

A log table contains data-locking information used by a server. The server tracks this information. Whenever a physical record is logged, information identifying the data being logged is placed in the log table. Normally, a set of records are logged and then locked as a set. However, a single record can also be locked when it is placed in the log table.

When using log tables, an application first logs all records to complete a transaction. The application then attempts to lock the logged set of records. If some of the logged resources cannot be locked, the lock fails and none of the resources are locked. Therefore, either all of the resources needed to complete a transaction are locked, or none of the resources are locked. This function cannot lock physical records that are exclusively locked by other applications.

The `fileHandle` is a valid file handle returned by a previous call to open, sopen, creat, or fileno.

When the `lockDirective` parameter specifies that the physical record should be locked when it is logged, the `timeoutLimit` parameter indicates how long the server should wait if it cannot lock the record immediately. The `timeoutLimit` parameter is specified in units of 1/18 of a second (0 means no wait).

See Also

ClearPhysicalRecord (page 452), ClearPhysicalRecordSet (page 454), LockPhysicalRecordSet (page 461), ReleasePhysicalRecord (page 475), ReleasePhysicalRecordSet (page 477)

OpenSemaphore

Opens the specified semaphore or creates it if it does not exist

Local Servers: blocking

Remote Servers: blocking

Classification: 3.x, 4.x, 5.x, 6.x

Service: Server-Based Synchronization

Syntax

```
#include <\nlm\nit\nwsync.h>

int OpenSemaphore (
   char   *semaphoreName,
   int     initialValue,
   long   *semaphoreHandle,
   WORD   *openCount);
```

Parameters

semaphoreName

　(IN) Specifies the string containing the name of the semaphore. (_MAX_SEMAPHORE_VALUE, defined in NWSYNC.H, is 128 characters, including the NULL terminator).

initialValue

　(IN) If semaphore does not yet exist, it is assigned this value (1 to 127).

semaphoreHandle

　(OUT) Receives semaphore handle.

openCount

　(OUT) Receives the number of processes that have the semaphore open.

Return Values

Decimal	Hex	Constant
0	(0x00)	ESUCCESS
254	(0xFE)	ERR_INVALID_SEMAPHORE_NAME_LENGTH
255	(0xFF)	ERR_INVALID_INITIAL_SEMAPHORE_VALUE

Remarks

For cross-platform functionality, call NWOpenSemaphore (page 371).

The `semaphoreValue` is ignored unless this call creates the semaphore (because it did not already exist). The value can range from 1 to 127, indicating that only 1 to 127 processes can access the network resource at a time. A call to SignalSemaphore increments this value. A call to WaitOnSemaphore decrements this value.

The `openCount` indicates the number of processes holding the semaphore open. A call to OpenSemaphore increments this value. A call to CloseSemaphore decrements this value.

The value returned in the `semaphoreHandle` parameter is the semaphore handle. The application must pass this value in calls to all other semaphore functions.

The application must pass either the name of an existing semaphore or the name of the new semaphore in the `semaphoreName` parameter. A semaphore name is an ASCIIZ string from 1 to 127 bytes long.

If the specified semaphore does not exist, this function creates and initializes the semaphore to `initialValue`.

The `openCount` parameter indicates the number of processes using the semaphore.

See Also

CloseSemaphore (page 455), ExamineSemaphore (page 456), SignalSemaphore (page 478), WaitOnSemaphore (page 479)

ReleaseFile

Unlocks the specified file in the log table but does not remove the file from the table

Local Servers: blocking

Remote Servers: blocking

Classification: 3.x, 4.x, 5.x, 6.x

Service: Server-Based Synchronization

Syntax

```
#include <\nlm\nit\nwsync.h>

int ReleaseFile (
   char   *fileName);
```

Parameters

fileName

(IN) Specifies the string containing filename with optional full path specification of the file to be unlocked (maximum 255 characters, including the NULL terminator).

Return Values

Decimal	Hex	Constant
0	(0x00)	ESUCCESS
255	(0xFF)	ERR_DOS_FILE_NOT_FOUND

Remarks

For cross-platform functionality, call NWReleaseFileLock2 (page 373).

This function allows the application to release files without corrupting the integrity of the personal file table.

The `fileName` parameter can specify either a file's complete pathname or a path relative to the current working directory (CWD).

SetCurrentNameSpace sets the name space that is used for parsing the path input to this function.

See Also

ClearFile (page 448), ClearFileSet (page 449), LockFileSet (page 458), LogFile (page 463), ReleaseFileSet (page 472)

ReleaseFileSet

Unlocks all files currently locked in the log table, but it does not remove them from the log table

Local Servers: blocking

Remote Servers: blocking

Classification: 3.x, 4.x, 5.x, 6.x

Service: Server-Based Synchronization

Syntax

```
#include <\nlm\nit\nwsync.h>

void ReleaseFileSet (void);
```

Remarks

For cross-platform functionality, call NWReleaseFileLockSet (page 375).

The files are not removed from the log table; therefore, the application can relock the files without relogging them.

This function is ignored if there are no locked files.

See Also

ClearFile (page 448), ClearFileSet (page 449), LockFileSet (page 458), LogFile (page 463), ReleaseFile (page 471)

ReleaseLogicalRecord

Unlocks a logical record in the log table but does not remove the record from the log table

Local Servers: blocking

Remote Servers: blocking

Classification: 3.x, 4.x, 5.x, 6.x

Service: Server-Based Synchronization

Syntax

```
#include <\nlm\nit\nwsync.h>

int ReleaseLogicalRecord (
   char   *logicalRecordName);
```

Parameters

logicalRecordName

   (IN) Specifies the string containing the record name of the record to be unlocked. (_MAX_LOGREC_NAME, defined in NWSYNC.H, is 128 characters, including the NULL terminator).

Return Values

Decimal	Hex	Constant
0	(0x00)	ESUCCESS
255	(0xFF)	ERR_NO_RECORD_FOUND

Remarks

For cross-platform functionality, call NWReleaseLogicalRecord (page 377).

A log table contains data-locking information used by a server. The server tracks this information. Whenever a logical record is logged, information identifying the data being logged is placed in the log table. Normally, a set of records are logged and then locked as a set. However, a record can also be locked when it is placed in the log table.

See Also

ClearLogicalRecord (page 450), ClearLogicalRecordSet (page 451), LockLogicalRecordSet (page 459), LogLogicalRecord (page 465), ReleaseLogicalRecordSet (page 474)

ReleaseLogicalRecordSet

Unlocks all logical records that are currently locked in the log table but does not remove them from the table

Local Servers: blocking

Remote Servers: blocking

Classification: 3.x, 4.x, 5.x, 6.x

Service: Server-Based Synchronization

Syntax

```
#include <\nlm\nit\nwsync.h>

void ReleaseLogicalRecordSet (void);
```

Remarks

For cross-platform functionality, call NWReleaseLogicalRecordSet (page 379).

The log table resides on the server and is associated exclusively with the requesting task. Consequently, the application can relock the logical record names without relogging them.

This function is ignored if there are no locked logical records.

See Also

ClearLogicalRecord (page 450), ClearLogicalRecordSet (page 451), LockLogicalRecordSet (page 459), LogLogicalRecord (page 465), ReleaseLogicalRecord (page 473)

ReleasePhysicalRecord

Unlocks the specified physical record currently locked in the log table but does not remove it from the log table

Local Servers: blocking

Remote Servers: blocking

Classification: 3.x, 4.x, 5.x, 6.x

Service: Server-Based Synchronization

Syntax

```
#include <\nlm\nit\nwsync.h>

int ReleasePhysicalRecord (
   int    fileHandle,
   long   recordStartOffset,
   long   recordLength);
```

Parameters

fileHandle

   (IN) Specifies the handle of the file containing the record to be unlocked.

recordStartOffset

   (IN) Specifies the offset within the file where the record begins.

recordLength

   (IN) Specifies the length of the record in bytes.

Return Values

Decimal	Hex	Constant
0	(0x00)	ESUCCESS
255	(0xFF)	ERR_NO_RECORD_FOUND
		No logged record was found with the specified description.

Remarks

For cross-platform functionality, call NWReleasePhysicalRecord (page 381).

The log table resides on the server and is associated exclusively with the requesting task. Since the function does not remove the physical record from the log table, the application can relock the physical record without relogging it.

The `fileHandle` value is returned by a previous open, sopen, creat, or fileno call.

See Also

ClearPhysicalRecord (page 452), ClearPhysicalRecordSet (page 454), LockPhysicalRecordSet (page 461), LogPhysicalRecord (page 467), ReleasePhysicalRecordSet (page 477)

ReleasePhysicalRecordSet

Unlocks all physical records currently locked in the log table but does not remove them from the log table

Local Servers: blocking

Remote Servers: blocking

Classification: 3.x, 4.x, 5.x, 6.x

Service: Server-Based Synchronization

Syntax

```
#include <\nlm\nit\nwsync.h>

void ReleasePhysicalRecordSet (void);
```

Remarks

For cross-platform functionality, call NWReleasePhysicalRecordSet (page 383).

The log table resides on the server and is associated exclusively with the requesting task. Since the function does not remove the physical records from the log table, the application can relock the physical records without relogging them.

This function is ignored if there are no locked physical records.

See Also

ClearPhysicalRecord (page 452), ClearPhysicalRecordSet (page 454), LockPhysicalRecordSet (page 461), LogPhysicalRecord (page 467), ReleasePhysicalRecord (page 475)

SignalSemaphore

Increments the value of the specified semaphore

Local Servers: nonblocking

Remote Servers: blocking

Classification: 3.x, 4.x, 5.x, 6.x

Service: Server-Based Synchronization

Syntax

```
#include <\nlm\nit\nwsync.h>

int SignalSemaphore (
   long    semaphoreHandle);
```

Parameters

semaphoreHandle

   (IN) Specifies the semaphore handle of an open semaphore.

Return Values

Decimal	Hex	Constant
0	(0x00)	ESUCCESS
1	(0x01)	ERR_INSUFFICIENT_SPACE
255	(0xFF)	ERR_INVALID_SEMAPHORE_HANDLE

Remarks

For cross-platform functionality, call NWSignalSemaphore (page 399).

An application must call this function when finished accessing the network resource associated with the semaphore. If there are processes waiting to use the semaphore (the semaphore value is negative), the first process in the queue is released (signaled).

An application should obtain a semaphore handle by calling OpenSemaphore.

See Also

CloseSemaphore (page 455), ExamineSemaphore (page 456), OpenSemaphore (page 469), WaitOnSemaphore (page 479)

WaitOnSemaphore

Decrements a semaphore value

Local Servers: blocking

Remote Servers: blocking

Classification: 3.x, 4.x, 5.x, 6.x

Service: Server-Based Synchronization

Syntax

```
#include <\nlm\nit\nwsync.h>

int WaitOnSemaphore (
   long   semaphoreHandle,
   WORD   timeoutLimit);
```

Parameters

semaphoreHandle

   (IN) Specifies the semaphore handle of an open semaphore.

timeoutLimit

   (IN) Specifies the timeout value.

Return Values

Decimal	Hex	Constant
0	(0x00)	ESUCCESS
1	(0x01)	SEMAPHORE_OVERFLOW
		Semaphore value was reached.
254	(0xFE)	ERR_TIMEOUT_FAILURE
		Timeout limit was reached before the semaphore was available.
255	(0xFF)	ERR_INVALID_SEMAPHORE_HANDLE

Remarks

For cross-platform functionality, call NWWaitOnSemaphore (page 401).

An application must call this function before accessing the network resource associated with the semaphore.

If the semaphore value is still greater than or equal to zero after the function decrements it, the application can access the associated resource. If the value is less than zero, the function queues the

application for the time interval specified in the `timeoutLimit` parameter. If the semaphore is incremented n + 1 times (where n is the negation of the semaphore value before the process called WaitOnSempahore), the process is restarted and WaitOnSempahore returns a value of 0. Otherwise, the process is restarted when the timeout expires and a value of 254 is returned.

An application can obtain a semaphore handle by calling OpenSemaphore.

The `timeoutLimit` indicates how long the server should wait if the semaphore value is negative. The `timeoutLimit` is specified in units of 1/18 of a second (0 means no wait).

WaitOnSemaphore temporarily disables the current connection number on which it is issued. This can pose a problem if the NLM thread group has set its current connection to that of a client before making the call. If this happens, the client is blocked out from getting file service requests until the unblock (that is, WaitOnSemaphore returns) occurs. If the delay is substantial, the client encounters the "Error sending/receiving on network" critical error.

To avoid this problem, an NLM should allocate connection numbers of its own when doing any network semaphore activity. A separate connection should be used for each thread group that uses network semaphores. Be cautious of using connection 0 for network semaphores. If a semaphore blocks, all other NLM applications in the system are prevented from doing any type of function that requires an enabled connection number.

See Also

CloseSemaphore (page 455), ExamineSemaphore (page 456), OpenSemaphore (page 469), SignalSemaphore (page 478)

Revision History

A

The following table outlines all the changes that have been made to the Single and Intra-File Management documentation (in reverse chronological order):

October 11, 2006	Updated ioctl (page 219) and pipe (page 232).
March 1, 2006	Updated format.
October 5, 2005	Transitioned to revised Novell documentation standards.
June 1, 2005	Modified the sample code for the fopen (page 274) function.
March 2, 2005	Modified the documentation for the flushall (page 273) function.
June 9, 2004	Added documentation for the Extended Attribute functions, which now accept UTF-8 strings. See NWOpenEAExt (page 172), NWReadEAExt (page 177), etc.
February 18, 2004	Added NetWare 6.5 information to DOSPresent (page 133).
July 30, 2003	Fixed typos.
June 2003	Fixed a typo in fgets. Changed all Pascal references to Delphi references.
October 2002	Fixed the Pascal syntax of the structures. Updated the documentation for sopen (page 238)
May 2002	Added descriptions of subfunctions to ioctl (page 219).
February 2002	Added another string example to printf (page 302) in the Remarks section.
	Updated the parameter descriptions of DOSChangeFileMode (page 124).
	Updated links.
October 2001	Updated Pascal syntaxes for PHYSICAL_LOCK (page 416) and PHYSICAL_LOCKS (page 418).
September 2001	Added support for NetWare 6.x to documentation.
	Added descriptions to graphics.
June 2001	Updated tables.
	Added 0x0001 as a Return Value for NWReadEA (page 174) and updated the Remarks section as to the significance of various return values.
February 2001	Added documentation for cancel (page 202) and setvbuf (page 321).
September 2000	Corrected the type for the size field in find_t (page 148).
	Created Chapter 26, "Server-Based Extended Attribute Structures," on page 439 and moved server-based EA structure definitions to that chapter.
	Removed a link to an irrelevant example from Chapter 12, "Extended Attribute Concepts," on page 149.
July 2000	Corrected sopen (page 238) to reflect the correct syntax and updated the Remarks section.

May 2000	Added documentation for DOSChangeFileMode (page 124), DOSRename (page 137), and DOSShutOffFloppyDrive (page 140). Added const to the parameters of several function definitions.
March 2000	Changed DFSSetDataSize (page 103) to be 5.x function. Also updated Remarks section to indicate this function works only on the Novell Storage System file system. Removed tmpfile example because it was inaccurate and misleading.
November 1999	Added descriptions for all structure fields in Synchronization Structures. Updated Remarks section of OpenEA (page 432) and added a NetWare 3.12 and 4.1 section that explains the possible return values if a bad server name or bad volume name are passed into this function. Changed nwshare.h to nwfattr.h in sopen (page 238). Added library information for each function.
September 1999	Added documentation for pipe (page 232).