www.novell.com/documentation

File System Services (64-Bit)

Developer Kit

August 2011

Novell

Legal Notices

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

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

Any products or technical information provided under this Agreement may be subject to U.S. export controls and the trade laws of other countries. You agree to comply with all export control regulations and to obtain any required licenses or classification to export, re-export, or import deliverables. You agree not to export or re-export to entities on the current U.S. export exclusion lists or to any embargoed or terrorist countries as specified in the U.S. export laws. You agree to not use deliverables for prohibited nuclear, missile, or chemical biological weaponry end uses. Please refer to http://www.novell.com/info/exports/ (http://www.novell.com/info/exports/) for more information on exporting Novell software. Novell assumes no responsibility for your failure to obtain any necessary export approvals.

Copyright © 2000-2011 Novell, Inc. All rights reserved. No part of this publication may be reproduced, photocopied, stored on a retrieval system, or transmitted without the express written consent of the publisher.

Novell, Inc.
1800 South Novell Place
Provo, UT 84606
U.S.A.
www.novell.com

Online Documentation: To access the online documentation for this and other Novell developer products, and to get updates, see developer.novell.com/ndk. To access online documentation for Novell products, see www.novell.com/documentation.

Novell Trademarks

For Novell trademarks, see the Novell Trademark and Service Mark list (http://www.novell.com/company/legal/trademarks/tmlist.html).

Third-Party Materials

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

Contents

About This Guide 7

1 Concepts 9

1.1 NetWare 6.5 SP1 Changes . . . 9
1.2 Pathname Resolution . . . 10
1.2.1 Fully Qualified Paths . . . 10
1.2.2 Relative Paths . . . 10
1.2.3 Namespace Considerations . . . 11
1.2.4 Wildcards . . . 12
1.3 Connection-Task-Transaction Model . . . 12
1.4 Keys . . . 13
1.5 Header Files . . . 13
1.6 Function Categories . . . 14
1.6.1 General File System . . . 14
1.6.2 Authorization . . . 14
1.6.3 Direct I/O . . . 14
1.6.4 Enumerate Directory . . . 15
1.6.5 Transaction . . . 15
1.6.6 Demigration . . . 15
1.7 OES 2 Functions . . . 15

2 Functions 17

2.1 Standard Functions . . . 18
zAbortXaction . . . 20
zAddTrustee . . . 21
zBeginTask . . . 22
zBeginXaction . . . 23
zClose . . . 25
zCommitXaction . . . 26
zCreate . . . 27
zDelete . . . 30
zDeleteTrustee . . . 32
zDIORead . . . 33
zDIOWrite . . . 35
zEndTask . . . 37
zEnumerate . . . 38
zFlush . . . 40
zGetEffectiveRights . . . 41
zGetFileMap . . . 42
zGetInfo . . . 44
zGetInfoByName . . . 46
zGetInheritedRightsMask . . . 48
zGetTrustee . . . 49
zInfoGetFileName . . . 50
zLink . . . 51
zLockByteRange . . . 53
zModifyInfo . . . 55
zModifyInfoByName . . . 57
zNewConnection . . . 59
zNewConnectionWithBlob . . . 60

zOpen . . . 61
zPrepareToCommitXaction . . . 63
zRead . . . 64
zRename . . . 66
zRootKey . . . 69
zSetEOF . . . 70
zSetInheritedRightsMask . . . 72
zUnlockByteRange . . . 73
zWildRead . . . 74
zWildRewind . . . 76
zWrite . . . 77
zZIDDelete . . . 79
zZIDDelete2 . . . 81
zZIDOpen . . . 83
zZIDRename . . . 85
zZIDRename2 . . . 87
2.2 Data Migration Functions . . . 89
DemigrateFunc_t . . . 90
zRegisterDemigrateFunction . . . 91
zUnregisterDemigrateFunction . . . 92

3 Structures 93

blockSize . . . 94
count . . . 95
dataStream . . . 96
deleted . . . 97
extAttr . . . 98
id . . . 99
names . . . 100
std . . . 101
storageUsed . . . 103
time . . . 104
zFILEMAP_ALLOCATION . . . 105
zFILEMAP_LOGICAL . . . 106
zFILEMAP_PHYSICAL . . . 107
zInfo_s . . . 108
zInfoB_s . . . 113
zMacInfo_s . . . 118
zPoolInfo_s . . . 119
zUnixInfo_s . . . 121
zVolumeInfo_s . . . 122

4 Values 125

4.1 Return Values . . . 126
4.1.1 zErrors (0-20013) . . . 127
4.1.2 Message, User Transaction, and Task Errors (20051-20070) . . . 128
4.1.3 eDirectory Errors (20090-20099) . . . 128
4.1.4 General File System Errors (20100-20105) . . . 128
4.1.5 Virtual File System Errors (20150-20159) . . . 129
4.1.6 General Storage System Errors (20200-20213) . . . 129
4.1.7 Admin Volume Errors (20250-20252) . . . 130
4.1.8 Beast Errors (20300-20312) . . . 130
4.1.9 Naming Errors (20400-20446) . . . 130
4.1.10 Name Typing Errors (20499) . . . 132
4.1.11 Rename Errors (20500-20512) . . . 132

4.1.12 Data Stream Errors (20550-20551) . . . 133
4.1.13 Semantic Agent Errors (20601-20602) . . . 133
4.1.14 Direct FS I/O & DIO Errors (20650-20655) . . . 133
4.1.15 Namespace Errors (20700-20705) . . . 133
4.1.16 Asynchronous Errors (20750-20751) . . . 134
4.1.17 Encrypted Volume Errors (20798-20799) . . . 134
4.1.18 Volume and Pool Errors (20800-20839) . . . 134
4.1.19 DSI and Adding Volumes Errors (20840-20849) . . . 136
4.1.20 Authorization Errors (20850-20871) . . . 136
4.1.21 NWCS Errors (20890-20898) . . . 137
4.1.22 Locking Errors (20900-20911) . . . 137
4.1.23 Unicode Errors (20950-20955) . . . 138
4.1.24 Link Errors (21000-21005) . . . 138
4.1.25 NLM Registration Errors (21050-21058) . . . 138
4.1.26 MASV Errors (21100) . . . 139
4.1.27 Modify Volume Info Errors (21150-21151) . . . 139
4.1.28 Feature Not Enabled Errors (21200-21215) . . . 139
4.1.29 User Space Restriction Errors (21300-21303) . . . 140
4.1.30 User Store Errors (21350-21351) . . . 140
4.1.31 Compression Manager Generated Errors (21400-21410) . . . 140
4.1.32 Directory Quota Errors (21500-21505) . . . 141
4.1.33 User Transaction Errors (21600-21606) . . . 141
4.1.34 Management File Errors (21700-21704) . . . 141
4.1.35 Data Migration Errors (21800-21803) . . . 142
4.1.36 Semantic Agent Errors (22000) . . . 142
4.1.37 CD-Specific Errors (22500-22512) . . . 142
4.1.38 DOSFAT/FAT32-Specific Errors (22600-22607) . . . 142
4.1.39 NSS Java Interface Errors (22900-22999) . . . 143
4.1.40 SMS Tape LSS Errors (23000-23024) . . . 143
4.1.41 NFS Gateway LSS Errors (23100-23199) . . . 143
4.1.42 Storage System B-Tree Errors (24800-24803) . . . 143
4.1.43 Storage System ZLOG Errors (24820-24831) . . . 143
4.1.44 ZVL Errors (24838-24839) . . . 144
4.1.45 ZFSVOL/ZLSSPOOL Volume Data Errors (24840-24849) . . . 144
4.1.46 Checkpoint Errors (24850-24853) . . . 145
4.1.47 Superblock Errors (24860-24868) . . . 145
4.1.48 Recovery Errors (24880) . . . 145
4.1.49 FSHooks Errors (24999) . . . 145
4.2 Common Values . . . 146
4.2.1 Effective Rights Values . . . 146
4.2.2 Features Values . . . 146
4.2.3 File Attributes Values . . . 148
4.2.4 File Type Values . . . 149
4.2.5 Get Info Mask Values . . . 150
4.2.6 Inherited Rights Mask Values . . . 151
4.2.7 Match Attributes Values . . . 152
4.2.8 Modify Info Mask Values . . . 153
4.2.9 Namespace ID Values . . . 154
4.2.10 Name Type Values . . . 155
4.2.11 Rename Flags Values . . . 155
4.2.12 Requested Rights Values . . . 155
4.2.13 Volume State Values . . . 157
4.3 Data Types . . . 157

Glossary **161**

A Revision History **167**

About This Guide

File System Services (64-Bit) runs on top of a message layer that DFS, Semantic Agents, and NSS developers use, and it hides partial path resolution and deprecated functions like reverse name mapping, which are needed for DFS and backward compatibility.

Are you wondering why Novell® is exposing yet another set of file system interfaces for NetWare®? Because this set of interfaces provides the following features:

- Handles files greater than 4GB
- Enables Unicode* file names
- Unifies interfaces
- Provides SMP-safe interfaces
- Improves interfaces for transactions
- Supports logical volumes, symbolic links, and junctions

This guide contains the following sections:

- "Concepts" on page 9
- "Functions" on page 17
- "Structures" on page 93
- "Values" on page 125
- "Glossary" on page 161
- "Revision History" on page 167

Audience

This guide is intended for CLib, NLM, and NKS (LibC) developers and provides complete access to the file system.

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 the File System Services (64-Bit) NDK page (http://developer.novell.com/ndk/fs64.htm).

Additional Information

For the related developer support postings for File System Services, see the Developer Support Forums (http://developer.novell.com/ndk/devforums.htm).

Documentation Conventions

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

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

1 Concepts

64-Bit File System Services (FS64) is a C language API that is exported by NetWare®. The FS64 functions are defined for your linker by an imports file. The interfaces are then bound when the coordinating NLM is loaded on NetWare.

No client or server component install is required to use the FS64 APIs. If NSS is running (and NSS is always running on NetWare 6.0 and later versions), FS64 APIs are available on the server.

No client-side APIs are provided because client-based services are used for server file I/O, such as Win32 CreateFile. Client-side 64-bit file I/O is supported where such APIs exist and where the client transport supports such functions.

Keep in mind the following features of File System Services (64-Bit):

- Hard linking to directories is not supported. You can hard link only to files.
- Only NSS volume names are supported. File System Services does not recognize eDirectory path names.
- Not all file attributes are supported.
- User transactions are supported only for zWrite and zSetEOF.
- The root key does not reference the root of a directory tree.
- Only some of the file types can be created.
- All time stamps passed as parameters are represented as UTC times (unsigned 64 bits, seconds relative to January 1, 1970 or microseconds from January 1, 1970).

This section contains the following topics:

- Section 1.1, “NetWare 6.5 SP1 Changes,” on page 9
- Section 1.2, “Pathname Resolution,” on page 10
- Section 1.3, “Connection-Task-Transaction Model,” on page 12
- Section 1.4, “Keys,” on page 13
- Section 1.5, “Header Files,” on page 13
- Section 1.6, “Function Categories,” on page 14
- Section 1.7, “OES 2 Functions,” on page 15

1.1 NetWare 6.5 SP1 Changes

Directory quotas changed in NetWare 6.5 SP1 to allow only two directory quota threads to run simultaneously. Other initialization requests for directory quotas are queued to run on one of the two threads. You will not notice any difference because the functions return with a success return value as

soon as the request is queued. It might take a while for the background threads to return the current amount in use value the first time a quota is set. However, this functionality keeps the system from going into high utilization.

This directory quota behavior change occurs whether you are using the 64-bit File System APIs, NLM and NetWare Libraries for C (CLib), or Libraries for C (LibC).

Also, any allocated search maps must be closed when they are no longer in use to avoid problems with reusing search maps.

1.2 Pathname Resolution

A path string identifies the final target file object.

The following sections provide examples and descriptions of various paths:

- Section 1.2.1, “Fully Qualified Paths,” on page 10
- Section 1.2.2, “Relative Paths,” on page 10
- Section 1.2.3, “Namespace Considerations,” on page 11
- Section 1.2.4, “Wildcards,” on page 12

If no context information is specified, the string must be a fully qualified path. If a path context was used, this path information is relative to the file object that was previously identified by the context information. However, if this path begins with a single backslash (\) character, the path is considered to be relative to the root of the volume identified by the key. (The above examples are using the path separators for the DOS and LONG namespaces. The path separators might vary for other namespaces.)

In the DOS and LONG namespaces, if the path begins with a volume name followed by a colon, the volume name must be local on the server parsing the path. The remaining portion of the path is considered to be relative to that volume.

AFPTCP works properly with the zAPIs in the Macintosh* and UNIX* namespaces. However, volume names need to be resolved in the LONG or DOS namespace.

1.2.1 Fully Qualified Paths

The following is an example of a fully qualified path that is relative to a volume name:

```
myvolume:users\brenda\myfile.dat
```

The first name (up to the colon) identifies a volume that must exist on the local server that is parsing the path. The remainder of the path is relative to the root of the specified volume. This type of path ignores any file object information identified by the key. It always specifies to begin parsing at the root of the identified volume.

1.2.2 Relative Paths

To be legal, the volume and starting directory information must have been previously identified in a relative path by using a key. The following is an example of a relative path:

```
brenda\myfile.dat
```

Volume Root Relative Path

The following is an example of a path that is relative to the root of a volume:

```
\users\brenda\myfile.dat
```

This type of path ignores any file object information identified by a context handle. It always specifies to begin parsing at the root of the identified volume.

Datastream Relative Path

The following is an example of a relative path that identifies a data stream that is named mydatastream of the `myfile.dat` file:

```
brenda\myfile.dat:mydatastream
```

Extended Attribute Relative Path

The following is an example of a relative path that identifies an extended attribute that is named myextendedattribute of the `myfile.dat` file:

```
brenda\myfile.dat::myextendedattribute
```

Relative Directory Sequences

The following is an example of a relative path that uses relative directory movement sequences. In this case, the neal directory must be a sibling directory of the brenda directory that contains the `hisfile.dat` file.

```
brenda\..\neal\hisfile.dat
```

1.2.3 Namespace Considerations

To tell the system which characters are legal, which characters are wild, or if the names are case insensitive, the namespace must be specified. Some other characteristics of the path are defaulted, but the following mode bits override any defaults:

zMODE_UTF8

The default character set for the path is Unicode*. However, by ORing zMODE_UTF8, UTF-8 character strings can be used.

zMODE_DELETED

Changes the name scanning to look for deleted files in the salvage system.

zMODE_LINK

If the last name in the path is a symbolic link, a junction, or URL link, the scanning does not follow the link but acts on the link object itself.

Path components are separated by the standard path separator characters for that namespace. Wildcards and relative directory sequences (such as \..\..) are also represented in the native client's namespace format.

In addition to the standard separators, the DOS and LONG namespaces recognize a colon (:) following a slash as being a data stream name separator. They recognize a double colon (::) following anywhere after a slash as being an extended attribute name separator.

1.2.4 Wildcards

The last leaf name in a path can optionally contain wildcards. Wildcards are supported only by the DOS and LONG namespaces and allow operations like zRename and zDelete to operate on multiple files with one request. The DOS and LONG namespaces share a common syntax, with the following wildcards:

Asterisk (simple Unicode or UTF-8 *)

An asterisk matches any character or sequence of characters except for periods. If a period is encountered in the filename, it is not matched by an asterisk. The matching pattern must contain an explicit period or another wildcard character that matches periods in order to proceed beyond the period. If the asterisk occurs at the end of a wildcard pattern or before a period, it matches a null string.

Question mark (simple Unicode or UTF-8 ?)

A question mark matches any single character except for a period. If a period is encountered in the filename, it is not matched by a question mark. The matching pattern must contain an explicit period or another wildcard character that matches the period in order to proceed beyond the period. If the question mark occurs at the end of a wildcard pattern or before a period, it matches a null string.

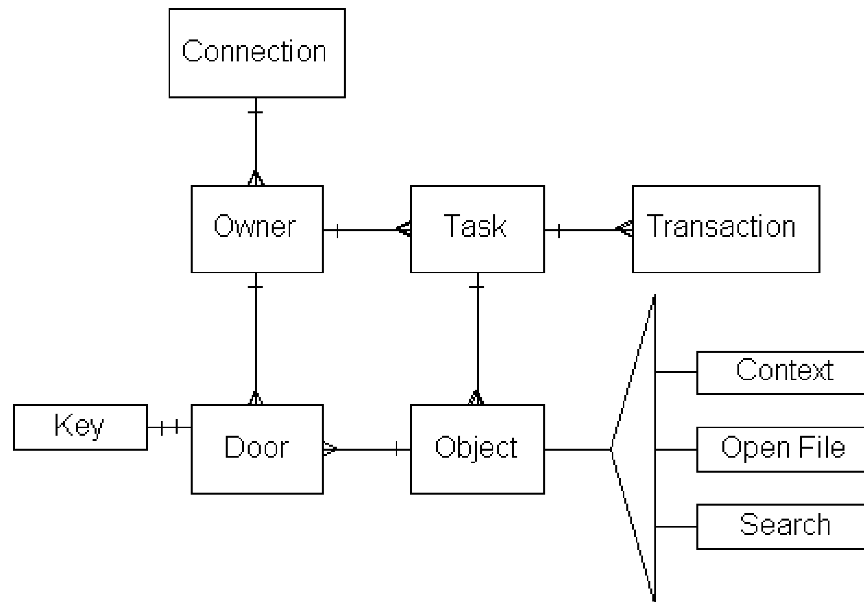
Period (simple Unicode or UTF-8 .)

A period matches a single period in the file name to be matched. If the period occurs at the end of the matching pattern, it also matches a null string

1.3 Connection-Task-Transaction Model

The following diagram shows how various transactions and tasks connect to the file system:

Figure 1-1 *Transaction Model*



The connection is an entity supplied by the OS. zRootKey creates the owner and the default task and transaction and returns the first key that has a NULL context (currently does not map to a directory or a file). New tasks can be created by calling zBeginTask.

If you want to use transactions, you must have transactions turned on for the volume. Every transaction is associated with a task, so a task ID must be supplied when a transaction is used. For some transactioned operations, the task ID is implied by the object being transactioned, such as for zRead and zWrite. When the key was created, a task ID was required and bound to the open file object during the zCreate or zOpen request. Because transactions are bound to a particular task, multiple tasks cannot use the same transaction.

The owner and door entities are transparent to the user and do not show in the functions.

1.4 Keys

To do anything with the file system, you must have a key. Keys are used to read and write files, open files, get information about files, scan directories, rename files, identify current directories, etc. Getting the first key can take some time because the user must be authenticated, but all subsequent keys are then ready to obtain and use.

A key is a non-persistent capability returned when opening a file object. However, some keys do not refer to file system objects, such as a Root key obtained from calling zRootKey (page 69). These types of keys cannot be identified by the value. The type of object a key refers to affects what operations can be performed on it.

Keys replace the traditional handles like file handles, context handles, and search maps. Keys also combine these roles with the role of the connection ID. Keys are not restricted to point to any particular NSS object but can point to any resource NSS chooses. The user must know what type of operations are allowed for the key requested. For example, if you have opened a file in Read-Only mode, write operations are denied. If you open a directory (or a file with extended attributes or multiple data streams), the directory can be scanned using the key with zWildRead (page 74). The underlying data structures keep track of the current location in the directory. If an illegal operation for a particular key is attempted, the functions return an error. When the key is destroyed by calling zClose (page 25) or some system operation, the resources associated with that key are released. A key can be destroyed by the owner (the client) of the key or the creator of the key (the file system—to force freeing resources).

A key is a 64-bit random number that hashes to a door, which points to a file handle structure that can represent an open file or a directory with a search map for enumerating the directory, a context for beginning a pathname lookup, or all of the above. When a file is forcibly closed by the system, the door can still exist but is now broken and cannot be used for further operations, which makes local and remote operations look the same.

1.5 Header Files

The following header files compose the File System Services header files and should be included after other Novell® header files:

- `zOmni.h` contains all the special typedefs used by the functions (BYTE, WORD, LONG, QUAD, etc.)
- `zError.h` contains the return values
- `zParams.h` contains all the #defines and structures used by the functions
- `zPublics.h` contains the function prototypes

1.6 Function Categories

File System Services functions are divided into the following categories:

- Section 1.6.1, "General File System," on page 14
- Section 1.6.2, "Authorization," on page 14
- Section 1.6.3, "Direct I/O," on page 14
- Section 1.6.4, "Enumerate Directory," on page 15
- Section 1.6.5, "Transaction," on page 15
- Section 1.6.6, "Demigration," on page 15

1.6.1 General File System

The following functions provide general file system functionality:

zBeginTask (page 22)
zClose (page 25)
zCreate (page 27)
zDelete (page 30)
zEndTask (page 37)
zFlush (page 40)
zGetFileMap (page 42)
zGetInfo (page 44)
zLink (page 51)
zModifyInfo (page 55)
zOpen (page 61)
zRead (page 64)
zRename (page 66)
zRootKey (page 69)
zSetEOF (page 70)
zWrite (page 77)

1.6.2 Authorization

The following functions assist you in providing file system rights to various users:

zAddTrustee (page 21)
zDeleteTrustee (page 32)
zGetInheritedRightsMask (page 48)
zGetTrustee (page 49)
zSetInheritedRightsMask (page 72)

1.6.3 Direct I/O

The following functions provide direct input and output capabilities:

zDIORead (page 33)
zDIOWrite (page 35)

zSetEOF (page 70)

1.6.4 Enumerate Directory

The following functions provide directory access using wildcards:

zWildRead (page 74)
zWildRewind (page 76)

1.6.5 Transaction

The following functions assist you with transaction tracking:

zAbortXaction (page 20)
zBeginXaction (page 23)
zCommitXaction (page 26)
zLockByteRange (page 53)
zPrepareToCommitXaction (page 63)
zUnlockByteRange (page 73)

1.6.6 Demigration

In addition to the standard set of 64-bit File System functions, there are two functions for demigrating files: zRegisterDemigrateFunction (page 91) and zUnregisterDemigrateFunction (page 92).

1.7 OES 2 Functions

The following functions are supported in the user space for Novell® Open Enterprise Server 2 (OES 2):

- zAddTrustee (page 21)
- zClose (page 25)
- zDelete (page 30)
- zDeleteTrustee (page 32)
- zEnumerate (page 38)
- zFlush (page 40)
- zGetEffectiveRights (page 41)
- zGetInfo (page 44)
- zGetInheritedRightsMask (page 48)
- zGetTrustee (page 49)
- zLink (page 51)
- zLockByteRange (page 53)
- zModifyInfo (page 55)
- zOpen (page 61)
- zRead (page 64)
- zRename (page 66)

- zRootKey (page 69)
- zSetEOF (page 70)
- zSetInheritedRightsMask (page 72)
- zUnlockByteRange (page 73)
- zWildRead (page 74)
- zWildRewind (page 76)
- zWrite (page 77)
- zZIDDelete (page 79)
- zZIDOpen (page 83)
- zZIDRename (page 85)

2 Functions

These functions are not supported on the existing NetWare® file system, but work only with the new NSS environment.

- Section 2.1, “Standard Functions,” on page 18
- Section 2.2, “Data Migration Functions,” on page 89

2.1 Standard Functions

This section contains the purpose, syntax, parameters, and return values for the following standard 64-bit File System functions:

- "zAbortXaction" on page 20
- "zAddTrustee" on page 21
- "zBeginTask" on page 22
- "zBeginXaction" on page 23
- "zClose" on page 25
- "zCommitXaction" on page 26
- "zCreate" on page 27
- "zDelete" on page 30
- "zDeleteTrustee" on page 32
- "zDIORead" on page 33
- "zDIOWrite" on page 35
- "zEndTask" on page 37
- "zEnumerate" on page 38
- "zFlush" on page 40
- "zGetEffectiveRights" on page 41
- "zGetFileMap" on page 42
- "zGetInfo" on page 44
- "zGetInfoByName" on page 46
- "zGetInheritedRightsMask" on page 48
- "zGetTrustee" on page 49
- "zInfoGetFileName" on page 50
- "zLink" on page 51
- "zLockByteRange" on page 53
- "zModifyInfo" on page 55
- "zModifyInfoByName" on page 57
- "zNewConnection" on page 59
- "zNewConnectionWithBlob" on page 60
- "zOpen" on page 61
- "zPrepareToCommitXaction" on page 63
- "zRead" on page 64
- "zRename" on page 66
- "zRootKey" on page 69
- "zSetEOF" on page 70
- "zSetInheritedRightsMask" on page 72
- "zUnlockByteRange" on page 73
- "zWildRead" on page 74

- "zWildRewind" on page 76
- "zWrite" on page 77
- "zZIDDelete" on page 79
- "zZIDDelete2" on page 81
- "zZIDOpen" on page 83
- "zZIDRename" on page 85
- "zZIDRename2" on page 87

zAbortXaction

Aborts a transaction.

Service: File System Services (64-Bit)

Syntax

```
#include <zPublics.h>

STATUS zAbortXaction(
   Key_t   key,
   Xid_t   xid);
```

Parameters

key

(IN) Specifies the owner of the transaction being aborted. Although any key derived from the same root key can be used for this function, it is easier to maintain the code if you reuse the same key.

xid

(IN) Specifies whether to abort this transaction. Any operations that were bound to this transaction are undone and the resources are returned to a state as if these operations had never been done. Changes made to the system that are not bound to the transaction are not affected.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, “Return Values,” on page 126 for a description).

Remarks

Aborting a transaction puts the system back into an equivalent state that existed before the transaction began. Even if this transaction is the child of a nested transaction, all locks held by the transaction are released.

See Also

zBeginXaction (page 23), zCommitXaction (page 26), zLockByteRange (page 53), zPrepareToCommitXaction (page 63), zUnlockByteRange (page 73)

zAddTrustee

Adds the list of trustees to those already assigned to the file.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zAddTrustee(
   Key_t          key,
   Xid_t          xid,
   const GUID_t  *trustee,
   NINT           rights);
```

Parameters

key

(IN) Specifies the file object where the trustee will be added, as returned by zOpen or zCreate.

xid

(IN) Specifies the transaction ID and binds the request to this transaction. If the requested action is not part of a transaction, use zNILXID.

trustee

(IN) Points to the trustee to add to the file object. The trustee is the eDirectory™ Globally Unique ID.

rights

(IN) Specifies the rights the specified trustee will have.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, “Return Values,” on page 126 for a description).

Remarks

Only the trustees that are not already assigned to the file are actually added.

See Also

zClose (page 25), zCreate (page 27), zDeleteTrustee (page 32), zGetInheritedRightsMask (page 48), zOpen (page 61), zGetTrustee (page 49), zSetInheritedRightsMask (page 72)

zBeginTask

Begins a new task.

Service: File System Services (64-Bit)

Syntax

```
#include <zPublics.h>

STATUS zBeginTask(
   Key_t   key,
   NINT    taskID,
   NINT   *retTaskID);
```

Parameters

key

(IN) Specifies the key that identifies the owner of the task.

taskID

(IN) Specifies the task ID number. If zero is passed in, the system assigns a task ID; otherwise, the user must supply a task ID in the range of 0-65,535.

retTaskID

(OUT) Points to a new task ID (a value greater than 65,535). When the task is ended, all the resources associated with the task are released.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, “Return Values,” on page 126 for a description).

Remarks

Resources such as open files and transactions are tied to a specific task.

See Also

zEndTask (page 37), zRootKey (page 69)

zBeginXaction

Begins a transaction.

Service: File System Services (64-Bit)

Syntax

```
#include <zPublics.h>

STATUS zBeginXaction(
   Key_t   key,
   NINT    taskID,
   Xid_t   parentXid,
   Xid_t  *retXid);
```

Parameters

key

(IN) Specifies the owner or key of the new transaction. We suggest that you use the same key to begin transactions and to commit or abort the transaction.

taskID

(IN) Specifies the transaction to bind the task to. If the task is released, all uncommitted transactions bound to the task are aborted.

parentXid

(IN) Specifies the parent ID. If set to zNILXID, a completely new transaction is started. However, if set to a transaction, that transaction becomes the parent to the new nested child transaction. When the child transaction commits, all of its locks are given to the parent transaction.

retXid

(OUT) Points to a transaction ID, which is passed to any operations that need to participate in the transaction. All operations that are bound to the transaction are processed as a group, either all complete or none complete.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, “Return Values,” on page 126 for a description).

Remarks

A transaction lets a set of operations be treated as a single atomic operation where either all of them are completed or none of them are completed.

A parent transaction cannot commit until all of its children and their descendants have committed. If a parent aborts, all of its children are aborted.

See Also

zAbortXaction (page 20), zCommitXaction (page 26), zLockByteRange (page 53), zPrepareToCommitXaction (page 63), zUnlockByteRange (page 73)

zClose

Closes a key.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zClose(
   Key_t   key);
```

Parameters

key

(IN) Specifies the object to close by the key that was returned by zOpen, zRootKey, or zCreate.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, “Return Values,” on page 126 for a description).

Remarks

If any specific actions are needed when the file is closed (such as deleting, flushing, or purging), those attributes should have been specified in the open request.

zClose invalidates the key and releases the object represented by the key. If the key is for an open file, it closes the file. If it is the key set up by zRootKey, not only is this instance of the connection closed, but all of the NSS resources associated with the connection are freed and any pending transactions are aborted.

See Also

zCreate (page 27), zRootKey (page 69), zOpen (page 61)

zCommitXaction

Commits a transaction.

Service: File System Services (64-Bit)

Syntax

```
#include <zPublics.h>

STATUS zCommitXaction(
   Key_t    key,
   Xid_t    xid);
```

Parameters

key

(IN) Specifies the key that identifies the owner of the transactions. Although any key derived from the same root key can be used for this function, it is easier to maintain the code if you use the same key that was used in zBeginXaction.

xid

(IN) Specifies the transaction. All operations bound to the transaction are then permanent.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, “Return Values,” on page 126 for a description).

Remarks

zCommitXaction does not return until all the operations bound to the transaction have been written to permanent storage. This does not mean the data has been written to the actual final fil. However, the data has been logged, so that if the system crashes, the recovery code can finish the operations. If this is a child of a nested transaction, all its locks are passed to its parent. If this is a root transaction, all of its locks are released before the call returns and after all the other operations have been committed.

See Also

zAbortXaction (page 20), zBeginXaction (page 23), zLockByteRange (page 53), zPrepareToCommitXaction (page 63), zUnlockByteRange (page 73)

zCreate

Creates and optionally opens a file object.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zCreate(
   Key_t        key,
   NINT         taskID,
   Xid_t        xid,
   NINT         nameSpace,
   const void  *path,
   NINT         fileType,
   QUAD         fileAttributes,
   NINT         createMode,
   NINT         requestedRights,
   Key_t       *retKey);
```

Parameters

key

(IN) Specifies who is creating the file. If the path is a relative path, it identifies the starting directory for resolving the pathname.

taskID

(IN) Specifies the task to which the returned key is bound.

xid

(IN) Specifies the transaction for which the file object is a part. The creation of the file object is part of this transaction. If the creation aborts, the file is deleted as if the creation never occurred. If there is no transaction, pass zNILXID.

nameSpace

(IN) Specifies the starting namespace for the path parameter. Only the long namespace should be used for normal file access.

In addition to the normal namespace identifiers (zNSPACE_DOS | zNSPACE_MAC | zNSPACE_UNIX | zNSPACE_LONG | zNSPACE_DATA_STREAM | zNSPACE_EXTENDED_ATTRIBUTE), the nameSpace parameter can be augmented by ORing the following mode bits:

- zMODE_UTF8

  The default character set for the path is Unicode. However, by ORing zMODE_UTF8, UTF-8 character strings can be used.

- zMODE_LINK

  If the last name in the path is a symbolic link, a junction, or a URL link, the scanning does not follow the link but acts on the link object itself.

path

(IN) Points to the path of the file object to be created. If the path is not fully qualified, the key must represent a directory for starting the path parsing. The default character set for the pathname is Unicode unless it is specified by the nameSpace parameter.

fileType

(IN) Specifies the type of the file to be created (see also "File Type Values" on page 149):

- zFILE_REGULAR for normal files and directories.
- zFILE_EXTENDED_ATTRIBUTE to create an extended attribute for a file.
- zFILE_NAMED_DATA_STREAM to create a data stream.

If the file object already exists and if the createMode specifies to reuse the existing file object, fileType is ignored.

fileAttributes

(IN) Specifies a bit mask that identifies specific file attributes to be associated with the newly created file (see "File Attributes Values" on page 148). If the file object already exists and if createMode specifies to reuse the existing file object, fileAttributes is ignored. The file attributes are completely independent from the authorization system and take precedence over the effective rights granted via that authorization system.

createMode

(IN) Specifies the actions to take place if the file object being created already exists. If the file object exists and createMode is zero, the create fails and one of the following error statuses is returned:

zCREATE_OPEN_IF_THERE	Opens the existing file, leaving its contents intact.
zCREATE_TRUNCATE_IF_THERE	Opens the existing file but truncates the data stream. The old contents are lost and are not saved in the file salvage system.
zCREATE_DELETE_IF_THERE	Truncates the existing file if the file has been opened with zRR_CANT_DELETE_WHILE_OPEN flag is set. Deletes the existing file and creates a new one from scratch. If salvage is enabled on the volume, the deleted file is moved to the file salvage system unless the zFA_IMMEDIATE_PURGE is set, in which case it is purged completely.

requestedRights

(IN) Specifies the rights to assign to retKey (see "Requested Rights Values" on page 155). If the file object is not being opened, requestedRights is ignored.

retKey

(IN/OUT) Points to whether a key should be returned:

NULL	The file should be created but not opened, and no key is returned.
non-NULL	A key is returned that gives the user access to the newly created and opened file object. Not only can this key can be used to read and write the created file, but it can be used to get and set information about this file by calling zGetInfo and zModifyInfo.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, “Return Values,” on page 126 for a description).

Remarks

zCreate can create many different types of file objects. Any file type specific information can be added later by either writing to the file object’s data stream or modifying the file object’s metadata.

Any necessary rights checking is done based on the requestedRights and key parameters.

If an error occurs, the file object is not created or opened.

See Also

zClose (page 25)

zDelete

Deletes one or more file names.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zDelete(
   Key_t        key,
   Xid_t        xid,
   NINT         nameSpace,
   const void  *path,
   NINT         matchAttribute,
   NINT         deleteFlags);
```

Parameters

key

(IN) Specifies who is deleting the file. If the path is a relative path, it also specifies the starting directory for resolving the pathname.

xid

(IN) Specifies the transaction. Any deletes that are part of a transaction are not completed until the transaction is committed. For no transaction, set to zNILXID.

nameSpace

(IN) Specifies the starting namespace for the path parameter. In addition to the normal namespace identifiers (zNSPACE_DOS | zNSPACE_MAC | zNSPACE_UNIX | zNSPACE_LONG | zNSPACE_DATA_STREAM | zNSPACE_EXTENDED_ATTRIBUTE), the nameSpace parameter can be augmented by ORing the following mode bits:

- zMODE_UTF8

  The default character set for the path is Unicode; however, by ORing zMODE_UTF8, UTF-8 character strings can be used.

- zMODE_DELETED

  Changes the name scanning to look for deleted files in the salvage system, which effectively purges the file.

- zMODE_LINK

  If the last name in the path is a symbolic link, a junction, or a URL link, the scanning does not follow the link but acts on the link object itself.

path

(IN) Points to the path of the file object to be deleted. If the path is not fully qualified, the key must represent a directory for starting the path parsing. The default character set for the pathname is Unicode unless it is specified by the nameSpace parameter.

matchAttributes

(IN) Specifies the attributes that the existing file must match for the file to be deleted (see "Match Attributes Values" on page 152).

deleteFlags

(IN) Specifies actions to take place on the file object being deleted.

zDELETE_PURGE_IMMEDIATE. Do not move the file into the salvage system. Purge it immediately so that it cannot be undeleted.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, "Return Values," on page 126 for a description).

Remarks

If the salvage feature is enabled, delete is more like a rename that places the file object in a state where its storage can be reused but it is no longer visible to naming, unless zMODE_DELETED is set in the nameSpaceID.

If wildcards are not being used and an error occurs, the file object is not deleted. If wildcards are being used, check the return values carefully to determine if no file objects were deleted, some file objects were deleted, or other file objects which matched the wildcards were deleted.

If the file object has zFA_IMMEDIATE_PURGE set in its file attributes, if the system default is to purge immediately, or if the file system is full, the file object is deleted completely. Otherwise, it is moved to the file salvage system, where it remains until it is salvaged, until a purge takes place, or until the space is needed for additional live files.

If the file object is a directory container and the directory container is not empty, the delete fails.When a file is deleted, all data streams and/or extended attributes associated with that file are also deleted.

If the salvage system is enabled, zDelete marks the file as zFILE_DELETED so that it can be recovered by zRename.

See Also

zCreate (page 27), zLink (page 51), zRename (page 66)

zDeleteTrustee

Deletes a single trustee from a file object.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zDeleteTrustee(
   Key_t          key,
   Xid_t          xid,
   const GUID_t  *trustee);
```

Parameters

key

(IN) Specifies the file object (as returned by zOpen or zCreate) from which the trustee will be deleted.

xid

(IN) Specifies the transaction to which the request is bound. If the requested action is not part of a transaction, pass zNILXID.

trustee

(IN) Points to the trustee to be deleted from the file object.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, “Return Values,” on page 126 for a description).

See Also

zAddTrustee (page 21), zClose (page 25), zCreate (page 27), zGetInheritedRightsMask (page 48), zOpen (page 61), zGetTrustee (page 49), zSetInheritedRightsMask (page 72)

zDIORead

Performs a direct read from an open file object that was opened by in Direct I/O mode.

Service: File System Services (64-Bit)

Syntax

```
#include <zPublics.h>

STATUS zDIORead(
   Key_t   key,
   QUAD    unitOffset,
   NINT    unitsToRead,
   ADDR    callBackContext,
   void  (*dioReadCallBack)(
      ADDR   reserved,
      ADDR   callBackContext,
      NINT   retStatus),
   void   *retBuffer);
```

Parameters

key

(IN) Specifies the file object to be read using Direct I/O (as returned by zOpen or zCreate with the zRR_DIO_MODE set in the requested rights field).

unitOffset

(IN) Specifies a starting location in the file object where the read operation will begin. A unit is 512 bytes in length. For example, unit 0 starts at logical offset 0, unit 1 starts at logical offset 512, etc.

unitsToRead

(IN) Specifies the number of 512-byte units to read from the file.

callBackContext

(IN) Specifies a user-provided cookie that is provided back to the user when the call back function is called. This value can be anything you want, if it fits in an ADDR type.

dioReadCallBack

(IN) Points to a callback function:

- NULL performs a synchronous read and waits until the read operation completes before it returns.
- non-NULL performs an asynchronous read, calls the specified function, and returns when the read is queued.

IMPORTANT: Because the callback function can be called from fast work-to-do routines and can be called from any processor, the callback function must be SMP enabled and non-blocking.

---

reserved

Is used internally to optimize performance and should be ignored.

callBackContext

(IN) Specifies the context for the callback function.

retStatus

(OUT) Specifies the return code of the callback function.

retBuffer

(OUT) Points to a buffer that contains the data that was read. You must allocate memory for the buffer, and it must be large enough to handle the requested number of units of data (each unit is 512 bytes).

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, “Return Values,” on page 126 for a description).
zERR_FILE_NOT_IN_DIO_MODE	The file is not in direct file mode.

Remarks

A direct read is one that does not go through an internal file system cache buffer. Instead, the data is read directly from the media into the provided buffer. In order to do a direct read, the file object must be opened by calling zOpen or zCreate with the requested rights set to zRR_DIO_MODE.

zDIORead reads the specified number of 512-byte units, starting at the specified logical unitNumber in the file identified by the key. If all units are not read, it fails.

zDIORead works with physical storage media of any sector size. However, it works optimally if the read operation begins on a sector boundary and if the read length is an integral number of sectors; otherwise, the file system must do some internal manipulation to read on non-sector boundaries. For example, if the media is made up of 512-byte sectors, all reads are optimal. However, if the media is made up of 1024 byte sectors, zDIORead works optimally by always reading an even number of units and by starting the reads on an even-numbered unit.

zDIORead bypasses the cache.

See Also

zCreate (page 27), zDIOWrite (page 35), zOpen (page 61)

zDIOWrite

Performs a direct write to an open file object that was opened in Direct I/O mode.

Service: File System Services (64-Bit)

Syntax

```
#include <zPublics.h>

STATUS zDIOWrite(
   Key_t      key,
   QUAD       unitOffset,
   NINT       unitsToWrite,
   ADDR       callBackContext,
   void     (*dioWriteCallBack)(
         ADDR   reserved,
         ADDR   callBackContext,
         NINT   retStatus),
 const void   *buffer);
```

Parameters

key

(IN) Specifies the file object to be written using Direct I/O as returned by zOpen or zCreate with the zRR_DIO_MODE set in requested rights.

unitOffset

(IN) Specifies a starting location in the file object where the write operation will begin. A unit is 512 bytes in length. For example, unit 0 starts at logical offset 0, unit 1 starts at logical offset 512, etc.

unitsToWrite

(IN) Specifies the number of 512-byte units to write to the file.

callBackContext

(IN) Specifies a user-provided cookie that is provided back to the user when the callback routine is called. This value can be anything you want, if it fits in an ADDR type.

dioReadCallBack

(IN) Points to a callback function:

- NULL performs a synchronous write and waits until the write operation completes before it returns.
- non-NULL performs an asynchronous write, calls the specified function, and returns when the write is queued.

IMPORTANT: Because the callback function can be called from fast work-to-do routines and can be called from any processor, the callback function must be SMP enabled and non-blocking.

---

reserved

Is used internally to optimize performance and should be ignored.

callBackContext

(IN) Specifies the context for the callback function.

retStatus

(OUT) Specifies the return code of the callback function.

buffer

(OUT) Points to a buffer that contains the data that is to be written to the file object. The buffer should be on a 512-byte boundary.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred. See Section 4.1, "Return Values," on page 126 for details.
zERR_FILE_NOT_IN_DIO_MODE	The file is not in direct file mode.

Remarks

A direct write is one that does not go through an internal file system cache buffer. Instead, the data is written directly to the media from the buffer provided by the caller. In order to perform a direct write, the file object must be opened by calling zOpen or zCreate with the zRR_DIO_MODE set in requested rights.

zDIOWrite writes the specified number of 512-byte units, starting at the specified logical unitNumber in the file identified by the key. If all of the units cannot be written, an error is returned.

zDIOWrite works with physical storage media of any sector size. However, it works optimally if the write operation begins on a sector boundary and if the read length is an integral number of sectors; otherwise, the file system must do some internal manipulation to write on non-sector boundaries. For example, if the media is made up of 512-byte sectors, all writes are optimal. However, if the media is made up of 1024-byte sectors, zDIOWrite works optimally by always writing an even number of units and by starting the writes on an even-numbered unit.

See Also

zCreate (page 27), zDIORead (page 33), zOpen (page 61)

zEndTask

Ends a task and cleans up all resources associated with the given task ID.

Service: File System Services (64-Bit)

Syntax

```
#include <zPublics.h>

STATUS zEndTask(
   Key_t   key,
   NINT    taskID);
```

Parameters

key

(IN) Specifies the owner of the task to be terminated.

taskID

(IN) Specifies the task to end.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, “Return Values,” on page 126 for a description).

Remarks

All resources associated with this task are released, which includes any keys that were opened using this task. Any transactions bound to this task that have not been committed are aborted.

See Also

zClose (page 25)

zEnumerate

Enumerates files, data streams, deleted files, or extended attributes contained by another file object and returns metadata information for the next entry that matches the search criteria.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zEnumerate (
   Key_t     key,
   QUAD      cookie,
   NINT      nameType,
   NINT      match,
   QUAD      getInfoMask,
   NINT      sizeRetGetInfo,
   NINT      infoVersion,
   void     *retGetInfo
   QUAD     *nextCookie);
```

Parameters

key

(IN) Specifies the directory type object (or file) that will be scanned for subcomponents. The key must be obtained by opening a file/directory with zRR_SCAN_ACCESS.

cookie

(IN) Specifies the starting position in the directory. Pass zero the first time you call zEnumerate. On subsequent calls, this value can be set to values previously returned from zEnumerate.

nameType

(IN) Specifies the type of name to be enumerated: zNTYPE_FILE, zNTYPE_DATA_STREAM, zNTYPE_EXTENDED_ATTRIBUTE, or zNTYPE_DELETED_FILE (see "Name Type Values" on page 155).

match

(IN) Specifies the attributes that the file objects must match (optional). File objects that do not satisfy the match criteria are skipped.

getInfoMask

(IN) Specifies the information to be returned (see "Get Info Mask Values" on page 150).

sizeRetGetInfo

(IN) Specifies the total size (in bytes) of the zInfo_s structure passed to retGetInfo, which includes the size of the variable size portion that is used to store any variable-sized pieces returned in the structure.

infoVersion

(IN) Specifies the version of the zInfo structure that is being used. For NetWare 6.x versions, there are two supported versions: zINFO_VERSION_A and zINFO_VERSION_B. zINFO_VERSION_B enables the ability to obtain directory quotas.

retGetInfo

(OUT) Is a caller-supplied structure—either zInfo_s (page 108) or zInfoB_s (page 113)—in which the requested information is returned. If NULL, no information about this file or directory object is returned.

nextCookie

(OUT) Points to a cookie that points to the next file object.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, “Return Values,” on page 126 for a description).
zERR_BUFFER_TOO_SMALL	The variable size portion of retGetInfo was not big enough to contain all of the requested optional data. Only the fixed data and any optional data that completely fit in the structure was returned.
zERR_NAME_NOT_FOUND_ IN_DIRECTORY	No more files were found in the directory.

Remarks

zEnumerate reads the next entry contained in the file object identified by key and returns information for the next contained file object that is associated with that directory entry. The key maintains a current directory position that is updated for each function iteration.

The cookie parameter can be used to reposition the cursor in the directory.

The type of file object being enumerated is dependent on the nameType associated with the key parameter. This nameType was associated with the key by calling zOpen.

zEnumerate can also return type-specific information about the file object. If zERR_BUFFER_TOO_SMALL is returned, all of the fixed length data was returned but some of the variable length was not returned because the buffer did not have enough room for all of the requested variable length data. Call zEnumerate again with a larger buffer to retrieve all of the information, ensuring that the value that is passed to the cookie parameter is the same value that was passed in the previous call.

All of the variable length data is indexed by the fixed portion of the structure. If the data is filled in, the index to that data is also filled in. If the data does not fit, the index to that data will be set to zero.

The returned data consists of the data specified by the bits set in getInfoMask.

See Also

zGetInfo (page 44), zModifyInfo (page 55), zOpen (page 61), zWildRead (page 74)

zFlush

Flushes (or writes to disk) all updates that were made to an open file object.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zFlush(
   Key_t   key);
```

Parameters

key

(IN) Specifies the file object to be flushed, as returned by zOpen or zCreate.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, “Return Values,” on page 126 for a description).

Remarks

If any file modifications are still in the cache, they are written to disk. zFlush blocks until the file flush is complete and then returns to the caller. zFlush should not be called as part of a transaction because the transaction could be aborted and might have extra work to do. The transaction automatically takes care of any flushing that needs to be done.

See Also

zClose (page 25), zCreate (page 27), zOpen (page 61), zRead (page 64), zSetEOF (page 70), zWrite (page 77)

zGetEffectiveRights

Returns the effective rights (inherited or explicit) an eDirectory object has to a given file or directory.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zGetEffectiveRights(
   Key_t    key,
   GUID_t   objId,
   NINT    *effectiveRights);
```

Parameters

key

(IN) Specifies the file or directory to which effective rights are to be returned, as created by zOpen or zCreate.

objId

(IN) Specifies the eDirectory object for which the effective rights are returned.

NOTE: This object ID is the value of the GUID eDirectory property for the object for which the effective rights are to be returned. This value is not the same ID that is used by traditional file system functions that return an object's effective rights.

effectiveRights

(OUT) Returns the effective rights for the specified object.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, “Return Values,” on page 126 for a description).

See Also

zAddTrustee (page 21), zCreate (page 27), zDeleteTrustee (page 32), zGetTrustee (page 49), zOpen (page 61)

zGetFileMap

Returns an extent list for a file object and indicates which bytes are allocated.

Service: File System Services (64-Bit)

Syntax

```
#include <zPublics.h>

STATUS zGetFileMap(
   Key_t    key,
   Xid_t    xid,
   QUAD     startingOffset,
   NINT     extentListFormat,
   NINT     bytesForExtents,
   void    *retExtentList,
   QUAD    *retEndingOffset,
   NINT    *retExtentListCount);
```

Parameters

key

(IN) Specifies the file whose file map will be retrieved, as returned by zOpen or zCreate.

xid

(IN) Specifies the transaction associated with this request. If the requested action is not part of a transaction, pass in zNILXID.

startingOffset

(IN) Specifies the starting byte offset in the file at which the extent list is to begin. To get an extent list of the whole file, pass zero. If multiple calls are being made, pass the value of retEndingOffset that was returned by a previous function iteration.

extentListFormat

(IN) Specifies what format the extent list information should be returned in:

zFILEMAP_ALLOCATION (page 105)	Returns a map of what portions of the file are physically allocated versus what portions are sparse. This is useful for operations such as backing up sparse files where only data that is actually allocated to a file object needs to be processed and all sparse holes in the file need to be ignored. If the file is not sparse, a single extent describing the entire file is returned.
zFILEMAP_LOGICAL (page 106)	Returns a map of the logical extents being used on the volume, which shows how fragmented a file is. This is useful for operations like defragging because it can determine how fragmented a file is. Each extent contains exactly one entry for every allocation fragment in the file, even if the extents are logically adjacent to each other.
zFILEMAP_PHYSICAL (page 107)	Returns a map of the physical blocks being used including what physical devices the blocks are on. This is useful for operations like backup where the backup devices can directly communicate with each other. Each extent contains information describing exactly where this file object's data is physically stored.

bytesForExtents

(IN) Specifies the size (in bytes) of the retExtentList buffer. As many extent list elements as will fit in the buffer are returned. Each extent list element contains information for one extent. The size of an extent list element varies based on the format of the extent being returned.

retExtentList

(IN/OUT) Points to a preallocated buffer, which is filled in with an array of extent list elements. The number of extents returned is based on the size of this buffer. If the passed buffer is large enough to hold only one extent, retEndingOffset must be passed as the starting offset to get additional extents until zERR_END_OF_FILE is returned.

retEndingOffset

(OUT) Points to the the ending byte offset at which the last returned extent ends, which can be passed to a subsequent function iteration to get additional information.

retExtentListCount

(OUT) Points to the number of extent list elements for which information was returned in the retExtentList buffer.

Return Values

zERR_END_OF_FILE	All extents have been returned (in which case retEndingOffset is irrelevant).

Remarks

To guarantee that the information returned by zGetFileMap does not change, the file should be opened with zRR_DENY_WRITE for the requested rights.

If all of the file returned extent list elements do not fit in the retExtentList buffer, zGetFileMap can be called multiple times to return the complete file map information.

Each extent element identifies a different fragment of the file on a physical device (like a hard disk). All fragments, including sparse ones, are represented by an extent.

See Also

zCreate (page 27), zOpen (page 61)

zGetInfo

Returns file status and metadata information for the given file object.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zGetInfo(
   Key_t    key,
   QUAD     getInfoMask,
   NINT     sizeRetGetInfo,
   NINT     infoVersion,
   void    *retGetInfo);
```

Parameters

key

(IN) Specifies the file (as returned by zOpen or zCreate) whose metadata information will be retrieved.

getInfoMask

(IN) Specifies the information to be returned (see "Get Info Mask Values" on page 150).

sizeRetGetInfo

(IN) Specifies the total size (in bytes) of the zInfo_s structure passed in as the retGetInfo parameter, which includes the size of the variable size portion that is used to store any variable-sized pieces returned in the structure.

infoVersion

(IN) Specifies the version of the zInfo structure that is being used. For NetWare 6.x versions, there are two supported versions: zINFO_VERSION_A and zINFO_VERSION_B. zINFO_VERSION_B enables the ability to obtain directory quotas.

retGetInfo

(OUT) Is a caller-supplied structure—either zInfo_s (page 108) or zInfoB_s (page 113)—in which the requested information is returned. If it isNULL, no information about this file or directory object is returned.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, "Return Values," on page 126 for a description).

Remarks

If zERR_BUFFER_TOO_SMALL is returned, all of the fixed-length data is returned, but some of the variable-length data might not be returned because the buffer did not have enough room for all of the requested variable-length data. Call zGetInfo again with a larger buffer.

All of the variable-length data is indexed from the fixed portion of the structures. If the data is filled in, the index to it is filled in. If the data does not fit, the index to it is set to zero.

See Also

zGetFileMap (page 42), zModifyInfo (page 55)

zGetInfoByName

Returns file status and metadata information for the given file object that is passed in by name.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zGetInfoByName(
   Key_t         key,
   NINT          nameSpace,
   const void   *path,
   QUAD          getInfoMask,
   NINT          sizeRetGetInfo,
   NINT          infoVersion,
   BOOL          useSnapShot,
   void         *retGetInfo);
```

Parameters

key

(IN) Specifies the file (as returned by zOpen or zCreate) whose metadata information will be retrieved.

nameSpace

(IN) Specifies the starting namespace for the path parameter. In addition to the normal namespace identifiers (zNSPACE_DOS | zNSPACE_MAC | zNSPACE_UNIX | zNSPACE_LONG | zNSPACE_DATA_STREAM | zNSPACE_EXTENDED_ATTRIBUTE), the nameSpace parameter can be augmented by ORing the following mode bits:

- zMODE_UTF8

  The default character set for the path is Unicode; however, by ORing zMODE_UTF8, UTF-8 character strings can be used.

- zMODE_DELETED

  Changes the name scanning to look for deleted files in the salvage system, which effectively purges the file.

- zMODE_LINK

  If the last name in the path is a symbolic link, a junction, or a URL link, the scanning does not follow the link but acts on the link object itself.

path

(IN) Points to the path of the file object to be created. If the path is not fully qualified, the key must represent a directory for starting the path parsing. The default character set for the pathname is Unicode unless it is specified by the nameSpace parameter.

getInfoMask

(IN) Specifies the information to be returned (see "Get Info Mask Values" on page 150).

sizeRetGetInfo

(IN) Specifies the total size (in bytes) of the zInfo_s structure passed in as the retGetInfo parameter, which includes the size of the variable size portion that is used to store any variable-sized pieces returned in the structure.

infoVersion

(IN) Specifies the version of the zInfo structure that is being used. For NetWare 6.x versions, there are two supported versions: zINFO_VERSION_A and zINFO_VERSION_B. zINFO_VERSION_B enables the ability to obtain directory quotas.

useSnapShot

(IN) Specifies whether to return information from the snapshot (if the volume has a snapshot and the file is a snapshot beast) or the original file.

retGetInfo

(OUT) Is a caller-supplied structure—either zInfo_s (page 108) or zInfoB_s (page 113)—in which the requested information is returned. If it is NULL, no information about this file or directory object is returned.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, "Return Values," on page 126 for a description).

Remarks

If zERR_BUFFER_TOO_SMALL is returned, all of the fixed-length data is returned, but some of the variable-length data might not be returned because the buffer did not have enough room for all of the requested variable-length data. Call zGetInfo again with a larger buffer.

All of the variable-length data is indexed from the fixed portion of the structures. If the data is filled in, the index to it is filled in. If the data does not fit, the index to it is set to zero.

See Also

zGetInfo (page 44), zModifyInfoByName (page 57), zInfoGetFileName (page 50)

zGetInheritedRightsMask

Returns the Inherited Rights Mask for the given file object.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zGetInheritedRightsMask(
   Key_t  key,
   LONG  *retInheritedRightsMask);
```

Parameters

key

(IN) Specifies the file object (as returned by Returned by zOpen or zCreate) whose inherited rights mask will be retrieved.

retInheritedRightsMask

(OUT) Points to the inherited rights mask for the file object (see “Inherited Rights Mask Values” on page 151).

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, “Return Values,” on page 126 for a description).

Remarks

The inherited rights mask controls which rights are inherited down the directory tree. Rights do not inherit across junctions, symbolic links, or URL links.

See Also

zAddTrustee (page 21), zClose (page 25), zCreate (page 27), zDeleteTrustee (page 32), zOpen (page 61), zGetTrustee (page 49), zSetInheritedRightsMask (page 72)

zGetTrustee

Reads all the trustees assigned to a file.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zGetTrustee(
   Key_t    key,
   NINT     startingIndex,
   GUID_t  *retTrustee,
   NINT    *retRights,
   NINT    *retNextIndex);
```

Parameters

key

(IN) Specifies the file object (as returned by zOpen or zCreate) whose trustees will be retrieved.

startingIndex

(IN) Specifies the starting index for reading the next set of trustees. Zero specifies to begin reading the trustees. Thereafter, it should be replaced by the index returned by retNextIndex.

retTrustee

(IN/OUT) Points to the buffer to return the trustee assigned to this file that corresponds to startIndex.

retRights

(OUT) Points to the rights associated with the returned trustee.

retNextIndex

(OUT) Points to the next index to be used in startingIndex. If this is set to negative one, all the trustees have been read.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, "Return Values," on page 126 for a description).

Remarks

zGetTrustee returns only the trustees assigned to a file that you are allowed to see. The index does not imply any order to the trustees. Instead, it is a simple way to keep track of where to begin again in the list of trustees. Only one trustee is returned at a time.

zInfoGetFileName

Returns a namespace-specific filename from the zInfo_s structure.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

unicode_t* zInfoGetFileName(
   zInfo_s  *info,
   NINT      nameSpace);
```

Parameters

info

(IN) Points to the zInfo_s structure, which contains detailed information about the file object.

nameSpace

(IN) Specifies the starting namespace for the path parameter. In addition to the normal namespace identifiers (zNSPACE_DOS | zNSPACE_MAC | zNSPACE_UNIX | zNSPACE_LONG | zNSPACE_DATA_STREAM | zNSPACE_EXTENDED_ATTRIBUTE), the nameSpace parameter can be augmented by ORing the following mode bits:

- zMODE_UTF8

  The default character set for the path is Unicode; however, by ORing zMODE_UTF8, UTF-8 character strings can be used.
- zMODE_DELETED

  Changes the name scanning to look for deleted files in the salvage system, which effectively purges the file.
- zMODE_LINK

  If the last name in the path is a symbolic link, a junction, or a URL link, the scanning does not follow the link but acts on the link object itself.

Return Values

Returns the file string on success or NULL (zero) on failure.

See Also

zGetInfoByName (page 46), zModifyInfoByName (page 57)

zLink

Creates a hard link to an already existing file object.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zLink(
   Key_t        key,
   Xid_t        xid,
   NINT         srcNameSpace,
   const void  *srcPath,
   NINT         srcMatchAttributes,
   NINT         dstNameSpace,
   const void  *dstPath
   NINT         renameFlags);
```

Parameters

key

(IN) Specifies who is creating the link on a file. If either srcpath or dstpath are relative paths, the key is the starting directory for pathname resolution.

xid

(IN) Specifies the transaction to which to bind operations. If the requested action is not part of a transaction, pass zNILXID.

srcNameSpace

(IN) Specifies the starting namespace for the path. In addition to the normal namespace identifiers (zNSPACE_DOS | zNSPACE_MAC | zNSPACE_UNIX | zNSPACE_LONG), nameSpace can be augmented by ORing the following mode bits:

- zMODE_UTF8

  The default character set for the path is Unicode. However, by ORing zMODE_UTF8, UTF-8 character strings can be used.

- zMODE_LINK

  If the last name in the path is a symbolic link, a junction, or a URL link, the scanning does not follow the link but acts on the link object itself.

srcPath

(IN) Points to the path of the file object to be accessed. If the path is not fully qualified, the key must represent a directory for starting the path parsing. Wildcards are not allowed in the path string. The default character set for the pathname is Unicode unless it is specified by srcNameSpace.

srcMatchAttributes

(IN) Specifies whether the file to be linked must match the given attributes (optional) (see "Match Attributes Values" on page 152).

dstNameSpace

(IN) Specifies the starting namespace for the path. In addition to the normal namespace identifiers (zNSPACE_DOS | zNSPACE_MAC | zNSPACE_UNIX | zNSPACE_LONG), nameSpace can be augmented by ORing the following mode bits:

- zMODE_UTF8 The default character set for the path is Unicode; however, by ORing zMODE_UTF8, UTF-8 character strings can be used.

dstPath

(IN) Points to the path of the target name of the file object to be linked. This cannot be NULL. If the path is not fully qualified, the key must represent a directory where the path parsing should begin. This path must resolve to a directory file that exists, and it must also contain a leaf (link) name that does not yet exist. The target directory must reside in the same volume as the source directory. The default character set for the pathname is Unicode unless it is specified by dstNameSpace.

renameFlags

(IN) Specifies a bit mask that identifies various modes to the rename function (see also "Rename Flags Values" on page 155):

zRENAME_ALLOW_RENAMES_TO_MYSELF Allows a file to be named to its same place without returning an error

zRENAME_THIS_NAME_SPACE_ONLY Specifies the source and destination namespaces must be the same

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, "Return Values," on page 126 for a description).

Remarks

No wildcards are allowed.

The srcPath must fully resolve to an existing file object, which cannot be a directory object, a data stream, or an extended attribute.

The hard link is created in the directory that is identified by the key and/or dstPath, with the new name being the leaf name specified in dstPath.

See Also

zRename (page 66)

zLockByteRange

Locks a byte range for the specified open file object.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zLockByteRange(
   Key_t    key,
   Xid_t    xid,
   NINT     mode,
   QUAD     startingOffset,
   QUAD     length,
   NINT     msecTimeout);
```

Parameters

key

(IN) Specifies the file (as returned by zOpen or zCreate) whose byte range is being locked.

xid

(IN) Specifies the transaction to which the lock is bound. If the requested action is not part of a transaction, pass zNILXID; the lock is then bound to the task specified by the key.

mode

(IN) Specifies the mode of the lock: either zLOCK_SHARED (read lock) or zLOCK_EXCLUSIVE (write lock).

startingOffset

(IN) Specifies the logical byte offset in the file where the lock begins.

length

(IN) Specifies the number of bytes to lock in the file. To lock all the bytes in a file, set startingOffset to 0 and length to zMAX_FILE_SIZE (0xffffffffffffffff: a 64 bit -1).

msecTimeout

(IN) Specifies the number of milliseconds that you are willing to wait to receive the lock. Otherwise, it immediately returns an error if the lock cannot be obtained.

Return Values

zOK	The operation completed successfully.
zERR_LOCK_WAITING	Did not get the lock in the specified amount of time.
non-0	An error occurred (see Section 4.1, “Return Values,” on page 126 for a description).

Remarks

zLockByteRange locks the number of bytes specified by length, starting at the startingOffset of the file identified by key. A lock bound to a transaction can only be unlocked by either ending or aborting the transaction. When the transaction holding the lock ends on a nested transaction, its parent inherits all the locks of the child. To prevent dirty reads of data and preserve reread semantics, locks are finally released when the root transaction commits.

See Also

zRead (page 64), zUnlockByteRange (page 73), zWrite (page 77)

zModifyInfo

Modifies file status and metadata information for the given file object.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zModifyInfo(
   Key_t        key,
   Xid_t        xid,
   QUAD         modifyInfoMask,
   NINT         sizeModifyInfo,
   NINT         infoVersion,
   const void  *modifyInfo);
```

Parameters

key

(IN) Specifies the file (as returned by zOpen or zCreate) whose metadata information will be modified.

xid

(IN) Specifies the transaction to which the request is bound. If there is no transaction, pass zNILXID.

modifyInfoMask

(IN) Specifies a bit mask that identifies which fields of modifyInfoMask are to be modified (see "Modify Info Mask Values" on page 153). If a bit is set in this bit mask, the corresponding field of modifyFileInfo is used to modify the file.

sizeModifyInfo

(IN) Specifies the total size (in bytes) of the zInfo_s structure passed in to modifyInfo, which includes the size of the variable size portion that is used to store any variable sized pieces returned in the structure.

infoVersion

(IN) Specifies the version of the zInfo structure that is being used. For NetWare 6.x versions, there are two supported versions: zINFO_VERSION_A and zINFO_VERSION_B. zINFO_VERSION_B enables the ability to set directory quotas.

modifyInfo

(IN) Is a caller-supplied structure—either zInfo_s (page 108) or zInfoB_s (page 113)—that contains the requested information.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, "Return Values," on page 126 for a description).

Remarks

Only the metadata corresponding to the bits set in modifyInfoMask is modified. For any fields that are namespace-specific (such as the name), they are modified only in the namespace identified by nameSpace.

If an error occurs, the metadata for the file is not modified.

See Also

zGetInfo (page 44), zGetFileMap (page 42)

zModifyInfoByName

Modifies file status and metadata information for the given file object that is passed in by name.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zModifyInfoByName(
   Key_t        key,
   Xid_t        xid,
   NINT         nameSpace,
   const void  *path,
   QUAD         modifyInfoMask,
   NINT         sizeModifyInfo,
   NINT         infoVersion,
   const void  *modifyInfo);
```

Parameters

key

(IN) Specifies the file (as returned by zOpen or zCreate) whose metadata information will be modified.

xid

(IN) Specifies the transaction to which the request is bound. If there is no transaction, pass zNILXID.

nameSpace

(IN) Specifies the starting namespace for the path parameter. In addition to the normal namespace identifiers (zNSPACE_DOS | zNSPACE_MAC | zNSPACE_UNIX | zNSPACE_LONG | zNSPACE_DATA_STREAM | zNSPACE_EXTENDED_ATTRIBUTE), the nameSpace parameter can be augmented by ORing the following mode bits:

- zMODE_UTF8

  The default character set for the path is Unicode; however, by ORing zMODE_UTF8, UTF-8 character strings can be used.

- zMODE_DELETED

  Changes the name scanning to look for deleted files in the salvage system, which effectively purges the file.

- zMODE_LINK

  If the last name in the path is a symbolic link, a junction, or a URL link, the scanning does not follow the link but acts on the link object itself.

path

(IN) Points to the path of the file object to be created. If the path is not fully qualified, the key must represent a directory for starting the path parsing. The default character set for the pathname is Unicode unless it is specified by the nameSpace parameter.

modifyInfoMask

(IN) Specifies a bit mask that identifies which fields of modifyInfoMask are to be modified (see "Modify Info Mask Values" on page 153). If a bit is set in this bit mask, the corresponding field of modifyFileInfo is used to modify the file.

sizeModifyInfo

(IN) Specifies the total size (in bytes) of the zInfo_s structure passed in to modifyInfo, which includes the size of the variable size portion that is used to store any variable sized pieces returned in the structure.

infoVersion

(IN) Specifies the version of the zInfo structure that is being used. For NetWare 6.x versions, there are two supported versions: zINFO_VERSION_A and zINFO_VERSION_B (which enables the ability to set directory quotas).

modifyInfo

(IN) Is a caller-supplied structure—either zInfo_s (page 108) or zInfoB_s (page 113)—that contains the requested information.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, "Return Values," on page 126 for a description).

Remarks

Only the metadata corresponding to the bits set in modifyInfoMask is modified. For any fields that are namespace-specific (such as the name), they are modified only in the namespace identified by nameSpace.

If an error occurs, the metadata for the file is not modified.

See Also

zGetInfo (page 44), zGetInfoByName (page 46), zInfoGetFileName (page 50), zModifyInfo (page 55)

zNewConnection

Creates a connection to NSS as the specified user.

Service: File System Services (64-Bit)

Version: OES 2 SP1

Syntax

```
#include <zPublics.h>

STATUS zNewConnection(
   Key_t            rootkey,
   const unicode_t  *fdn)
```

Parameters

rootkey

(IN) Specifies the authentication key that is obtained from zRootKey(). Once you complete with the key, release it by calling zClose. When the connection key is closed, the NSS resources and keys associated with that connection are released.

const unicode_t *fdn

(IN) Points to the fully distinguished name of the user to which you want to establish the connection.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, “Return Values,” on page 126 for a description).

Remarks

zNewConnection works only with zRootKey.

See Also

zClose (page 25), zCreate (page 27), zDIORead (page 33), zDIOWrite (page 35), zRead (page 64), zSetEOF (page 70), zWrite (page 77)

zNewConnectionWithBlob

Creates a connection to NSS as the specified user along with the user specified data in the connection.

Service: File System Services (64-Bit)

Version: OES 2 SP3

Syntax

```
#include <zPublics.h>

STATUS zNewConnection(
   Key_t            rootkey,
   const unicode_t  *fdn)
   NINT            length,
   const BYTE      *blob);
```

Parameters

rootkey

(IN) Specifies the authentication key that is obtained from zRootKey(). Once you complete with the key, release it by calling zClose. When the connection key is closed, the NSS resources and keys associated with that connection are released.

const unicode_t *fdn

(IN) Points to the fully distinguished name of the user to which you want to establish the connection.

length

(IN) Specifies the length (in bytes) of the blob data.

blob

(IN) User interested data, the same is passed to the audit events.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, “Return Values,” on page 126 for a description).

Remarks

zNewConnection works only with zRootKey.

See Also

zClose (page 25), zCreate (page 27), zDIORead (page 33), zDIOWrite (page 35), zRead (page 64), zSetEOF (page 70), zWrite (page 77)

zOpen

Opens a file object.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zOpen(
   Key_t        key,
   NINT         taskID,
   NINT         nameSpace,
   const void  *path,
   NINT         requestedRights,
   Key_t       *retKey);
```

Parameters

key

(IN) Specifies who is opening the file. If the path is a relative path, the key identifies the starting directory for pathname resolution.

taskID

(IN) Specifies the task the returned key will be bound to.

nameSpace

(IN) Specifies the starting namespace for the path parameter. Only the long namespace should be used for normal file access.

In addition to the normal namespace identifiers (zNSPACE_DOS | zNSPACE_MAC | zNSPACE_UNIX | zNSPACE_LONG | zNSPACE_DATA_STREAM | zNSPACE_EXTENDED_ATTRIBUTE), nameSpace can be augmented by ORing the following mode bits:

- zMODE_UTF8

  The default character set for the path is Unicode. However, by ORing zMODE_UTF8, UTF-8 character strings can be used.

- zMODE_DELETED

  Changes the name scanning to look for deleted files in the salvage system, which effectively purges the file.

- zMODE_LINK

  If the last name in the path is a symbolic link, a junction, or a URL link, the scanning does not follow the link but acts on the link object itself.

path

(IN) Points to the path of the file object to be opened. If the path is not fully qualified, the key must represent a directory where the path parsing should begin. The default character set for the pathname is Unicode unless it is specified by nameSpace.

requestedRights

(IN) Specifies the rights that are requested for this instance of an open file structure (see "Requested Rights Values" on page 155).

retKey

(OUT) Points to a key that gives the user access to the open file object. This key can be used to open data streams, extend attributes relative to this file, and get information about this file (in addition to reading and writing).

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, "Return Values," on page 126 for a description).

Remarks

If an error occurs, the file object is not opened. Before opening the file object, the object is queried to find out if it can be opened. A file object has the right to refuse to be opened.

Any necessary rights checking is done based on the requestedRights and key parameters.

The requestedRights are associated with the returned key.

See Also

zClose (page 25), zCreate (page 27), zDIORead (page 33), zDIOWrite (page 35), zRead (page 64), zSetEOF (page 70), zWrite (page 77)

zPrepareToCommitXaction

Prepares the system to commit a transaction.

Service: File System Services (64-Bit)

Syntax

```
#include <zPublics.h>

STATUS zPrepareToCommitXaction(
   Key_t   key,
   Xid_t   xid);
```

Parameters

key

(IN) Specifies the owner of the transaction being prepared for commit. Alhough any key derived from the same root key can be used for this operation, it is easier to maintain the code if you use the same key that was used in zBeginXaction.

xid

(IN) Specifies the transaction to prepare so that it can be committed or aborted if requested.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, “Return Values,” on page 126 for a description).

Remarks

zPrepareToCommitXaction is similar to zCommitXaction, except it can still be aborted and is needed only to coordinate multiple transactions. In the simplest case, a transaction coordinator tells all the transactions it is coordinating to prepare. After every transaction has responded, they can all be committed.

See Also

zAbortXaction (page 20), zBeginXaction (page 23), zCommitXaction (page 26)

zRead

Reads from an open file object.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zRead(
   Key_t    key,
   Xid_t    xid,
   QUAD     startingOffset,
   NINT     bytesToRead,
   void    *retBuffer,
   NINT    *retBytesRead);
```

Parameters

key

(IN) Specifies the file object (as returned by zOpen or zCreate) that is to be read.

xid

(IN) Specifies the transaction to which the request is bound. If the requested action is not part of a transaction, pass zNILXID.

startingOffset

(IN) Specifies which logical byte offset to begin reading in the file.

bytesToRead

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

retBuffer

(OUT) Points to the data that was read.

retBytesRead

(OUT) Points to the number of bytes that were read.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, "Return Values," on page 126 for a description).

Remarks

zRead reads the number of bytes specified by bytesToRead bytes, starting at the startingOffset from the file identified by key. The actual number of bytes read might be smaller than the requested number of bytes to be read. The retBuffer is supplied by the caller and must be large enough to hold the requested data.

See Also

zClose (page 25), zCreate (page 27), zFlush (page 40), zOpen (page 61), zWrite (page 77)

zRename

Renames one or more file objects.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zRename(
   Key_t        key,
   Xid_t        xid,
   NINT         srcNameSpace,
   const void  *srcPath,
   NINT         srcMatchAttributes,
   NINT         dstNameSpace,
   const void  *dstPath,
   NINT         renameFlags);
```

Parameters

key

(IN) Specifies who is renaming the files. If one or both paths are relative pathnames, it also specifies the starting directory for pathname resolution.

xid

(IN) Specifies the transaction to which the operation is bound. If the requested action is not part of a transaction, pass zNILXID.

srcNameSpace

(IN) Specifies the starting namespace for the path parameter. In addition to the normal namespace identifiers (zNSPACE_DOS | zNSPACE_MAC | zNSPACE_UNIX | zNSPACE_LONG | zNSPACE_DATA_STREAM | zNSPACE_EXTENDED_ATTRIBUTE), nameSpace can be augmented by ORing the following mode bits:

- zMODE_UTF8

  The default character set for the path is Unicode. However, by ORing zMODE_UTF8, UTF-8 character strings can be used.

- zMODE_DELETED

  Changes the name scanning to look for deleted files in the salvage system, which effectively purges the file.

- zMODE_LINK

  If the last name in the path is a symbolic link, a junction, or a URL link, the scanning does not follow the link but acts on the link object itself.

srcPath

(IN) Points to the path of the file object to be renamed. If the path is not fully qualified, the key must represent a directory for starting the path parsing. The default character set for the pathname is Unicode unless it is specified by srcNameSpace.

srcMatchAttributes

(IN) Specifies the attributes that the file(s) to be renamed must match (optional) (see "Match Attributes Values" on page 152).

dstNameSpace

(IN) Specifies the starting namespace for the path parameter. In addition to the normal namespace identifiers (zNSPACE_DOS | zNSPACE_MAC | zNSPACE_UNIX | zNSPACE_LONG | zNSPACE_DATA_STREAM | zNSPACE_EXTENDED_ATTRIBUTE), nameSpace can be augmented by ORing the following mode bits:

- zMODE_UTF8

  The default character set for the path is Unicode. However, by ORing zMODE_UTF8, UTF-8 character strings can be used.

- zMODE_LINK

  If the last name in the path is a symbolic link, a junction, or a URL link, the scanning does not follow the link but acts on the link object itself.

dstPath

(IN) Points to the target path and name of the file object to be renamed. This cannot be NULL.Wildcards are allowed in the path string. This path must resolve to a directory file that exists, and it must also contain a leaf (link) name that does not yet exist.The target directory must reside in the same volume as the source directory. The default character set for the pathname is Unicode unless it is specified by dstNameSpace.

renameFlags

(IN) Specifies a bit mask that identifies various modes to the rename function (see also "Rename Flags Values" on page 155):

- zRENAME_ALLOW_RENAMES_TO_MYSELF allows a file to be named to its same place without returning an error.
- zRENAME_THIS_NAME_SPACE_ONLY specifies the source and destination namespaces must be the same.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, "Return Values," on page 126 for a description).

Remarks

The return values indicates that none of the file objects were renamed or that some file objects were renamed while others were not.

If zRENAME_THIS_NAME_SPACE_ONLY is set in renameFlags, zRename renames a single file object (no wildcards allowed) in only the specified srcNameSpace and the file object cannot be moved to a different directory path. The name of the file object remains unchanged in all namespaces other than the specified srcNameSpace.

If zRENAME_THIS_NAME_SPACE_ONLY is not set, zRename is much more flexible. The srcPath and dstPath can include wildcards, and they can be in different directories, provided that they are on the same volume. With wildcards, multiple files objects can be renamed with a single iteration of

zRename.The nameSpace identifies the primary namespace for which the destination names apply, but the file objects are renamed in all namespaces on the volume. An appropriate destination name in the other namespaces is generated from dstPath.

zRename can be called to salvage files that have been deleted. When a file is deleted, its name type is changed from zNTYPE_FILE to zNTYPE_DELETED_FILE. zRename can then change the name type back to zNTYPE_FILE, thus making the file reappear in the directory.

See Also

zLink (page 51)

zRootKey

Authenticates a connection with NSS to establish the first key for use with subsequent requests.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zRootKey(
   NINT    connectionID,
   Key_t  *retRootKey);
```

Parameters

connectionID

(IN) Specifies the connection that you want to authenticate with NSS. If you want to mask the eDirectory connection number, use 0xFFFF with the connection handle. If you are using the real connection number, you do not need to mask any of the bits.

retRootKey

(OUT) Points to the authentication key, which can be used with other interfaces to get additional keys and access NSS objects like directories, files, and search maps. When you are finished with the key, release it by calling zClose. When the connection key is closed, the NSS resources and keys associated with that connection are released.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, “Return Values,” on page 126 for a description).

Remarks

The key returned by zRootKey represents the connection to NSS. This key has a different set of legal operations from a file system object key. For example, zGetInfo is not supported for root keys.

When the key is closed, all the NSS resources associated with the connection are released. Because this key can be used to obtained other keys, those keys are also automatically closed when this key is closed.

The strength of the authentication largely determines how secure the rest of the zAPIs are.

If an error occurs, the connection is not authenticated and no key is returned.

See Also

zClose (page 25), zCreate (page 27), zOpen (page 61)

zSetEOF

Sets either the logical or physical end of file for a data stream.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zSetEOF(
   Key_t   key,
   Xid_t   xid,
   QUAD    dataSizeBytes,
   NINT    setSizeFlags);
```

Parameters

key

(IN) Specifies the file (as returned by zOpen or zCreate) whose end of file will be changed.

xid

(IN) Specifies the transaction to which the request is bound. If the requested action is not part of a transaction, pass zNILXID.

dataSizeBytes

(IN) Specifies the logical byte offset in the file object at which the new end of file is to be set, which applies only to the data stream identified by key.

setSizeFlags

(IN) Specifies a bit mask that identifies various modes to the set data size function:

zSETSIZE_NON_ SPARSE_FILE	If this bit is not set and if the file size is being expanded, the file is expanded in a sparse manner with no data blocks being physically allocated to the file. If this bit is set and if the file size is being expanded, the file is expanded in a non-sparse manner, with data blocks being physically allocated to the file. The data block is also zeroed unless the zSETSIZE_NO_ZERO_FILL bit is set.
zSETSIZE_NO_ ZERO_FILL	If this bit is not set and if the file size is being expanded in a non-sparse manner, zero-filled data blocks are physically written to the file starting at the current end of file and ending at the new end of file. If this bit is set and if the file size is being expanded in a non-sparse manner, the new disk blocks allocated to the file are not zero-filled. The blocks are physically allocated to the file, but they are not initialized. This bit is allowed only from the NLM interface and cannot be set by an NCP that expands the file.

zSETSIZE_UNDO_ ON_ERR	If the file size is being expanded and if an error occurs anywhere during the expansion process, this bit is checked to determine the next action. If this bit is set, an error is returned and the file is restored to its original size as if no expansion took place. If this bit is not set, an error is returned and any partially completed expansion is left as part of the file.
zSETSIZE_ PHYSICAL_ONLY	If this bit is set, only the physical end of file is changed; the logical end of file is untouched. If the file is expanded with this option set, it is done in a non-sparse manner as if the zSETSIZE_NON_SPARSE_FILE bit were set. If the file is truncated with this bit set, the file is physically truncated, but the logical end of file remains unchanged.
zSETSIZE_ LOGICAL_ONLY	If this bit is set, zSetEOF does not touch physical storage, either to expand it or to truncate it. It only changes the logical end of file and leaves the physical storage that is allocated to the file unchanged.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, “Return Values,” on page 126 for a description).

Remarks

zSetEOF modifies the data size (end of file) for the given open file object. If the file has more than one data stream, only the size of the data stream identified by the key is modified. All other data streams of the same file object are left unmodified. In order to modify the size of more than one data stream, each one must be opened independently and this function must be called once for each data stream.

If the new size is smaller than the existing size, the data at the end of the data stream is truncated and discarded.

If the new size is larger than the existing size, the data size is expanded. The file can be expanded in either a sparse or non-sparse manner. If setSizeFlags has the zSETSIZE_NON_SPARSE_FILE bit set, the file will be expanded in a non-sparse manner. If zSETSIZE_NO_ZERO_FILL is set, this non-sparse expanding is done without writing zeros. Otherwise, the file is expanded by physically writing zeros from the current size up to the new end of file.

If the zSETSIZE_NON_SPARSE_FILE bit is not set, the file will be expanded in a sparse manner with no data being actually written to the file. If a read operation is performed on sparse areas of a file, the read buffer is filled in with zeros.

If an error occurs during the process of expanding a file with zero-filled data, one of two possible actions takes place depending on the value of the zSETSIZE_UNDO_ON_ERR bit. If this bit is set, any partial expansions are unexpanded and the file is restored to its original state before the call was made. If this bit is not set and an error occurs partway through the expansion process, the partial expansion is left as part of the file with the rest of the expansion aborted.

See Also

zClose (page 25), zCreate (page 27), zFlush (page 40), zOpen (page 61), zRead (page 64), zWrite (page 77)

zSetInheritedRightsMask

Sets the inherited rights mask for the given file object.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zSetInheritedRightsMask(
   Key_t   key,
   Xid_t   xid,
   LONG    inheritedRightsMask);
```

Parameters

key

(IN) Specifies the file object (as returned by zOpen or zCreate) whose inherited rights mask will be set.

xid

(IN) Specifies the transaction to bind this request to. If the requested action is not part of a transaction, pass zNILXID.

inheritedRightsMask

(IN) Specifies the value the file object's inherited rights mask should be set to (see "Inherited Rights Mask Values" on page 151).

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, "Return Values," on page 126 for a description).

Remarks

The inherited rights mask controls which rights are inherited down the directory tree when resolving a pathname.

See Also

zAddTrustee (page 21), zClose (page 25), zCreate (page 27), zDeleteTrustee (page 32), zGetInheritedRightsMask (page 48), zGetTrustee (page 49), zOpen (page 61)

zUnlockByteRange

Unlocks a byte range for the specified open file object.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zUnlockByteRange(
   Key_t    key,
   Xid_t    xid,
   QUAD     startingOffset,
   QUAD     length);
```

Parameters

key

(IN) Specifies the file (as returned by zOpen or zCreate) whose byte range is being unlocked.

xid

(IN) Is currently not implemented. Pass zNILXID.

startingOffset

(IN) Specifies the logical byte offset in the file where the lock begins.

length

(IN) Specifies the length (in bytes) of the lock to be released. The xid, startingOffset, and length must match to release the lock.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, "Return Values," on page 126 for a description).

Remarks

Only locks not bound to a transaction can be unlocked by calling zUnlockByteRange, which prevents dirty reads and rereads for transactions.

See Also

zLockByteRange (page 53), zOpen (page 61), zWrite (page 77)

zWildRead

Enumerates files, data streams, deleted files, or extended attributes contained by another file object and returns metadata information for the next entry that matches the search criteria.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zWildRead(
   Key_t          key,
   NINT           characterCode,
   NINT           nameType,
   const void    *pattern,
   NINT           match,
   QUAD           getInfoMask,
   NINT           sizeRetGetInfo,
   NINT           infoVersion,
   void          *retGetInfo);
```

Parameters

key

(IN) Specifies the directory type object (or file) that will be scanned for subcomponents. The key must be obtained by opening a file/directory with zRR_SCAN_ACCESS.

characterCode

(IN) Specifies the character code for the pattern: zPFMT_UNICODE or zPFMT_UTF8.

nameType

(IN) Specifies the type of name to be enumerated with this wild read: zNTYPE_FILE, zNTYPE_DATA_STREAM, zNTYPE_EXTENDED_ATTRIBUTE, or zNTYPE_DELETED_FILE (see "Name Type Values" on page 155).

pattern

(IN) Points to a wildcard pattern to be used in searching for the next name in the directory container. If it is NULL, the next name in the directory container is returned. Otherwise, this search pattern (with optional wildcards) is used to locate the next name for which an entry is returned.The pattern uses the rules for the namespace passed to zOpen.

match

(IN) Specifies the attributes that the file objects must match (optional, see "Match Attributes Values" on page 152). File objects which do not satisfy the match criteria are skipped.

getInfoMask

(IN) Specifies the information to be returned (see "Get Info Mask Values" on page 150).

sizeRetGetInfo

(IN) Specifies the total size (in bytes) of the zInfo_s structure passed to retGetInfo, which includes the size of the variable size portion that is used to store any variable-sized pieces returned in the structure.

infoVersion

(IN) Specifies the version of the zInfo structure that is being used. For NetWare 6.x versions, there are two supported versions: zINFO_VERSION_A and zINFO_VERSION_B. zINFO_VERSION_B enables the ability to obtain directory quotas.

retGetInfo

(OUT) Is a caller-supplied structure—either zInfo_s (page 108) or zInfoB_s (page 113)—in which the requested information is returned. If NULL, no information about this file or directory object is returned.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, "Return Values," on page 126 for a description).

Remarks

zWildRead reads the next entry contained in the file object identified by the key and returns information for the next contained file object associated with that directory entry. The key maintains a current directory position (which is a handle returned by calling zOpen) that is updated for each function iteration.

The type of file object being enumerated is dependent on the nameType associated with the key. This nameType was associated with the key in the call to zOpen.

The search pattern identified by the path is a pattern (with or without wildcard sequences) used in selecting the next contained file object. A NULL pattern selects the next contained file object regardless of its name. Pattern matching does not apply the UNIX or MAC namespaces.

zWildRead can also return type-specific information about the file object. If zERR_BUFFER_TOO_SMALL is returned, all of the fixed-length data was returned but some of the variable-length was not returned because the buffer provided did not have enough room for all of the requested variable-length data. Call zWildRead again with a larger buffer.

All of the variable-length data is indexed by the fixed portion of the structures. If the data is filled in, the index to it is filled in. If the data would not fit, the index to it is set to zero.

The returned data consists of the data specified by the bits set in getInfoMask.

See Also

zGetInfo (page 44), zModifyInfo (page 55), zOpen (page 61), zWildRewind (page 76)

zWildRewind

Sets the current zWildRead position in an open container file object back to the beginning of the container.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zWildRewind(
   Key_t   key);
```

Parameters

key

(IN) Specifies the file or directory whose search map is to be reset.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, “Return Values,” on page 126 for a description).

Remarks

zWildRewind is used to reset the current read position in a container file object back to the beginning of the container, as if zOpen had just been called.

If an error occurs, the position is not reset.

See Also

zOpen (page 61), zWildRead (page 74)

zWrite

Writes to an open file object.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zWrite(
   Key_t        key,
   Xid_t        xid,
   QUAD         startingOffset,
   NINT         bytesToWrite,
   const void  *buffer,
   NINT        *retBytesWritten);
```

Parameters

key

(IN) Specifies the file object (as returned by zOpen or zCreate) being written.

xid

(IN) Specifies the transaction associated with this request. If the requested action is not part of a transaction, pass zNILXID.

startingOffset

(IN) Specifies the logical byte offset in the file object at which the write is to begin. If this is beyond the current end of file, the file is sparsely extended and leaves a hole in the file between the old EOF and this startingOffset.

bytesToWrite

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

buffer

(IN) Points to the data to be written to the file object.

retBytesWritten

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

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, “Return Values,” on page 126 for a description).

Remarks

zWrite writes the number of specified bytes from the buffer, starting at the startingOffset to the open file object identified by the key. The actual number of bytes written can be smaller than the requested number of bytes to be written.

See Also

zClose (page 25), zCreate (page 27), zFlush (page 40), zOpen (page 61), zRead (page 64), zSetEOF (page 70)

zZIDDelete

Deletes the file designated by the specified ZID.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zZIDDelete(
   Key_t        key,
   Xid_t        xid,
   VolumeID_t  *volumeID,
   Zid_t        zid,
   NINT         deleteFlags);
```

Parameters

key

(IN) Specifies who is opening the file. If the volume ID is not supplied, the key identifies which volume to use.

xid

(IN) Specifies that any deletes that are part of the given transaction are not completed until the transaction is committed. For no transaction, set to zNILXID.

volumeID

(IN) Points to the GUID of the volume on which to search for the ZID. If NULL, the key parameter is used to identify the volume.

zid

(IN) Specifies the ZID of the file to be opened.

deleteFlags

(IN) Specifies a bit mask that identifies specific actions that are to take place on the file object that is being deleted.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, “Return Values,” on page 126 for a description).

Remarks

zZIDDelete works only with files and directories. It does not work with extended attributes or data streams.

If the salvage feature is enabled, delete is more like a rename operation that puts the file object in a state where its storage can be reused but the file object itself is no longer visible to naming (unless zMODE_DELETED is set in the nameSpaceID).

See Also

zOpen (page 61), zZIDOpen (page 83), zZIDRename (page 85), zZIDRename2 (page 87)

zZIDDelete2

Deletes the file designated by the specified ZID.

Service: File System Services (64-Bit)

Version: OES 2 SP3

Syntax

```
#include <zPublics.h>

STATUS zZIDDelete2(
   Key_t        key,
   Xid_t        xid,
   VolumeID_t  *volumeID,
   Zid_t        zid,
   NINT         nameSpace
   NINT         deleteFlags);
```

Parameters

key

(IN) Specifies who is opening the file. If the volume ID is not supplied, the key identifies which volume to use.

xid

(IN) Specifies that any deletes that are part of the given transaction are not completed until the transaction is committed. For no transaction, set to zNILXID.

volumeID

(IN) Points to the GUID of the volume on which to search for the ZID. If NULL, the key parameter is used to identify the volume.

zid

(IN) Specifies the ZID of the file to be opened.

nameSpace

(IN) Specifies the starting namespace for the path parameter. In addition to the normal namespace identifiers (zNSPACE_DOS | zNSPACE_MAC | zNSPACE_UNIX | zNSPACE_LONG), the nameSpace parameter can be augmented by ORing the following mode bits:

- zMODE_UTF8

  The default character set for the path is Unicode; however, by ORing zMODE_UTF8, UTF-8 character strings can be used.

- zMODE_DELETED

  Changes the name scanning to look for deleted files in the salvage system, which effectively purges the file.

- zMODE_LINK

  If the last name in the path is a symbolic link, a junction, or a URL link, the scanning does not follow the link but acts on the link object itself.

deleteFlags

(IN) Specifies a bit mask that identifies specific actions that are to take place on the file object that is being deleted.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, "Return Values," on page 126 for a description).

Remarks

zZIDDelete2 works only with files and directories. It does not work with extended attributes or data streams.

If the salvage feature is enabled, delete is more like a rename operation that puts the file object in a state where its storage can be reused but the file object itself is no longer visible to naming (unless zMODE_DELETED is set in the nameSpaceID).

See Also

zOpen (page 61), zZIDOpen (page 83), zZIDRename (page 85), zZIDRename2 (page 87)

zZIDOpen

Opens a file object by using its ZID, instead of a name.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zZIDOpen(
   Key_t        key,
   NINT         taskID,
   NINT         nameSpace,
   VolumeID_t  *volumeID,
   Zid_t        zid,
   NINT         requestedRights,
   Key_t       *retKey);
```

Parameters

key

(IN) Specifies who is opening the file. If the volume ID is not supplied, the key identifies which volume to use.

taskID

(IN) Specifies the task the returned key will be bound to.

nameSpace

(IN) Specifies the namespace for the open file so that zEnumerate and zWildRead know which namespace to use when returning file names.

volumeID

(IN) Points to the GUID of the volume on which to search for the ZID. If NULL, the key parameter is used to identify the volume.

zid

(IN) Specifies the ZID of the file to be opened.

requestedRights

(IN) Specifies the rights that are requested for this instance of an open file structure (see "Requested Rights Values" on page 155).

retKey

(OUT) Points to a key that gives the user access to the open file object. This key can be used to open data streams, extend attributes relative to this file, and get information about this file (in addition to reading and writing). It can also be used to enumerate any additional data streams, extended attributes, or files associated with the file object.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, “Return Values,” on page 126 for a description).

Remarks

zZIDOpen works only with files and directories. It does not work with extended attributes or data streams. This function is useful for opening a file when you have problems representing the name.

If an error occurs, the file object is not opened. Before opening the file object, the object is queried to find out if it can be opened. A file object has the right to refuse to be opened.

Any necessary rights checking is done based on the requestedRights and key parameters.

The requestedRights are associated with the returned key.

See Also

zOpen (page 61), zZIDDelete (page 79), zZIDDelete2 (page 81), zZIDRename (page 85), zZIDRename2 (page 87)

zZIDRename

Renames the file object designated by the specified ZID.

Service: File System Services (64-Bit)

Version: OES 2

Syntax

```
#include <zPublics.h>

STATUS zZIDRename(
   Key_t        key,
   Xid_t        xid,
   VolumeID_t  *srcVolumeID,
   Zid_t        srcZid,
   Zid_t        dstZid,
   NINT         dstNameSpace,
   const void  *dstPath,
   NINT         renameFlags);
```

Parameters

key

(IN) Specifies who is renaming the file. If the volume ID is not supplied, the key identifies which volume to use. If the destination path is a relative path, the key is used to find the starting directory.

xid

(IN) Specifies that any deletes that are part of the given transaction are not completed until the transaction is committed. For no transaction, set to zNILXID.

srcVolumeID

(IN) Points to the GUID of the volume on which to search for the ZID. If NULL, the key parameter is used to identify the volume.

srcZid

(IN) Specifies the ZID of the file to be opened.

destZid

(IN) Specifies the ZID for the renamed file.

dstNameSpace

(IN) Specifies the starting namespace for the path parameter.

In addition to the normal namespace identifiers (zNSPACE_DOS | zNSPACE_MAC | zNSPACE_UNIX | zNSPACE_LONG | zNSPACE_DATA_STREAM | zNSPACE_EXTENDED_ATTRIBUTE), nameSpace can be augmented by ORing the following mode bits:

- zMODE_UTF8

  The default character set for the path is Unicode. However, by ORing zMODE_UTF8, UTF-8 character strings can be used.

- zMODE_LINK

  If the last name in the path is a symbolic link, a junction, or a URL link, the scanning does not follow the link but acts on the link object itself.

dstPath

(IN) Specifies the target path and name of the file objects. This path must resolve to a directory file that exists, and it must also contain a leaf (link) name which does not yet exist. A major restriction on this path is that the target directory must reside in the same volume as the source directory. The default character set for the pathname is Unicode (unless modified by the dstNameSpace parameter). Cannot be NULL.

renameFlags

(IN) Specifies a bit mask that identifies specific modes with the rename function as follows:

zRENAME_ALLOW_RENAMES_TO_MYSELF	Allows a file to be named to its same place without getting an error message.
zRENAME_THIS_NAME_SPACE_ONLY	Specifies that the source and destination namespaces must be the same.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, “Return Values,” on page 126 for a description).

Remarks

zZIDRename works only with files and directories. It does not work with extended attributes or data streams.

See Also

zRename (page 66), zZIDDelete (page 79), zZIDDelete2 (page 81), zZIDOpen (page 83)

zZIDRename2

Renames the file object designated by the specified ZID.

Service: File System Services (64-Bit)

Version: OES 2 SP3

Syntax

```
#include <zPublics.h>

STATUS zZIDRename2(
   Key_t        key,
   Xid_t        xid,
   VolumeID_t  *srcVolumeID,
   Zid_t        srcZid,
   NINT         srcNameSpace,
   Zid_t        dstZid,
   NINT         dstNameSpace,
   const void  *dstPath,
   NINT         renameFlags);
```

Parameters

key

(IN) Specifies who is renaming the file. If the volume ID is not supplied, the key identifies which volume to use. If the destination path is a relative path, the key is used to find the starting directory.

xid

(IN) Specifies that any deletes that are part of the given transaction are not completed until the transaction is committed. For no transaction, set to zNILXID.

srcVolumeID

(IN) Points to the GUID of the volume on which to search for the ZID. If NULL, the key parameter is used to identify the volume.

srcZid

(IN) Specifies the ZID of the file to be opened.

srcNameSpace

(IN) Specifies the starting namespace for the path parameter. In addition to the normal namespace identifiers (zNSPACE_DOS | zNSPACE_MAC | zNSPACE_UNIX | zNSPACE_LONG | zNSPACE_DATA_STREAM | zNSPACE_EXTENDED_ATTRIBUTE), the nameSpace parameter can be augmented by ORing the following mode bits:

- zMODE_UTF8

  The default character set for the path is Unicode; however, by ORing zMODE_UTF8, UTF-8 character strings can be used.

- zMODE_DELETED

  Changes the name scanning to look for deleted files in the salvage system, and effectively salvages the file.

- zMODE_LINK

  If the last name in the path is a symbolic link, a junction, or a URL link, the scanning does not follow the link but acts on the link object itself.

destZid

(IN) Specifies the ZID for the renamed file.

dstNameSpace

(IN) Specifies the starting namespace for the path parameter.

In addition to the normal namespace identifiers (zNSPACE_DOS | zNSPACE_MAC | zNSPACE_UNIX | zNSPACE_LONG), nameSpace can be augmented by ORing the following mode bits:

- zMODE_UTF8

  The default character set for the path is Unicode. However, by ORing zMODE_UTF8, UTF-8 character strings can be used.

- zMODE_LINK

  If the last name in the path is a symbolic link, a junction, or a URL link, the scanning does not follow the link but acts on the link object itself.

dstPath

(IN) Specifies the target path and name of the file objects. This path must resolve to a directory file that exists, and it must also contain a leaf (link) name which does not yet exist. A major restriction on this path is that the target directory must reside in the same volume as the source directory. The default character set for the pathname is Unicode (unless modified by the dstNameSpace parameter). Cannot be NULL.

renameFlags

(IN) Specifies a bit mask that identifies specific modes with the rename function as follows:

zRENAME_ALLOW_RENAMES_TO_MYSELF	Allows a file to be named to its same place without getting an error message.
zRENAME_THIS_NAME_SPACE_ONLY	Specifies that the source and destination namespaces must be the same.

Return Values

zOK	The operation completed successfully.
non-0	An error occurred (see Section 4.1, “Return Values,” on page 126 for a description).

Remarks

zZIDRename2 works only with files and directories. It does not work with extended attributes or data streams.

See Also

zRename (page 66), zZIDDelete (page 79), zZIDDelete2 (page 81), zZIDOpen (page 83)

2.2 Data Migration Functions

At any given time, Real Time Data Migration (RTDM) functions must be called by only one user within a system.

This section contains the purpose, syntax, parameters, and return values for the following RTDM functions:

- "DemigrateFunc_t" on page 90
- "zRegisterDemigrateFunction" on page 91
- "zUnregisterDemigrateFunction" on page 92

DemigrateFunc_t

Is called to demigrate files.

Service: File System Services (64-Bit)

Syntax

```
#include <zMigrate.h>

STATUS (*demigrateFunction) (
   key_t   key,
   QUAD    offset,
   QUAD    length);
```

Parameters

key

(IN) Specifies that read/write access is to be granted to the migrated file. This parameter can be used to write back to the file the data that has been migrated.

offset

(IN) Specifies the offset in the file where the user is starting to access the file. This parameter helps determine the minimum amount of data that must be migrated. However, the user can choose to migrate more data.

length

(IN) Specifies the length of the data that must be demigrated, starting from the offset parameter.

Return Values

zOK	The operation completed successfully.
zERR_DEMIGRATE_ ALREADY_REGISTERED	A demigration function has been registered. Only one demigrator can reside in the system at a time.

Remarks

When you request that NSS access the data portion of a migrated file, NSS calls this demigration function. After this demigration function returns, NSS expects the data to be demigrated and begins accessing the data.

To demigrate a file, call zWrite (page 77) to write the demigrated data back to the data stream. After this demigrated function returns, the key parameter is cleaned up.

See Also

zRegisterDemigrateFunction (page 91)

zRegisterDemigrateFunction

Registers a function to demigrate files.

Service: File System Services (64-Bit)

Syntax

```
#include <zMigrate.h>

STATUS zRegisterDemigrateFunction(
   DemigrateFunc_t   demigrateFunction);
```

Parameters

demigrateFunction

(IN) Specifies the demigrate function. NSS calls this function when you ask it to access the data for a migrated file.

Return Values

zOK	The operation completed successfully.
zERR_DEMIGRATE_ ALREADY_REGISTERED	A demigration function has been registered. Only one demigrator can reside in the system at a time.

Remarks

When you request that NSS access the data portion of a migrated file, NSS calls the demigration function. After the demigration function returns, NSS expects the data to be demigrated and begins accessing the data.

The key parameter that is passed to the demigration function can be used with the File System Services functions to demigrate a file. Call zWrite (page 77) to write the demigrated data back to the data stream. After the demigrated function returns, the key parameter is cleaned up.

See Also

zUnregisterDemigrateFunction (page 92)

zUnregisterDemigrateFunction

Cleans up the function that is used to demigrate files.

Service: File System Services (64-Bit)

Syntax

```
#include <zMigrate.h>

STATUS zUnregisterDemigrateFunction(
   void);
```

Return Values

zOK	The demigration function was cleared successfully.
non-0	An error occurred (see Section 4.1, "Return Values," on page 126 for a description).

Remarks

When the demigration function is deregistered, all active keys that were previously passed to the demigration function become inactive. If the system is in the middle of demigrating a file, further attempts to write to the data stream of that file will fail.

See Also

zRegisterDemigrateFunction (page 91)

3 Structures

This documentation describes the fields of the following File System Services structures:

- "blockSize" on page 94
- "count" on page 95
- "dataStream" on page 96
- "deleted" on page 97
- "extAttr" on page 98
- "id" on page 99
- "names" on page 100
- "std" on page 101
- "storageUsed" on page 103
- "time" on page 104
- "zFILEMAP_ALLOCATION" on page 105
- "zFILEMAP_LOGICAL" on page 106
- "zFILEMAP_PHYSICAL" on page 107
- "zInfo_s" on page 108
- "zInfoB_s" on page 113
- "zMacInfo_s" on page 118
- "zPoolInfo_s" on page 119
- "zUnixInfo_s" on page 121
- "zVolumeInfo_s" on page 122

blockSize

Is used by zInfo_s (page 108) and contains information about the size of the block.

Service: File System Services (64-Bit)

Defined In: zParams.h

Syntax

```
struct                                  /* zGET_BLOCK_SIZE */
{
   LONG   size;
   LONG   sizeShift;
} blockSize;
```

Fields

size

Specifies the logical size (which is always a power of 2 and a multiple of the underlying storage pool allocation unit size) of the file object's basical allocation blocks.

sizeShift

Specifies the shift factor that is used in working with the size of the block. If size is 2 raised to the nth power, the value of sizeShift is *n*. If 1 is shifted left by this value, the resultant value is the same value as size.

count

Is used by zInfo_s (page 108) and contains information about the number of file handles and directory entries for the file object.

Service: File System Services (64-Bit)

Defined In: zParams.h

Syntax

```
struct                              /* zGET_COUNTS */
{
   LONG   open;
   LONG   hardLink;
} count;
```

Fields

open

Specifies the number of open file handles that currently exist for this file object.

hardLink

Specifies the number of directory entries that point to this file object, including the original parent directory. If no additional hard links have been created for this file object, this field is set to 1.

dataStream

Is used by zInfo_s (page 108) and contains information about the number of data streams that are owned by the file object.

Service: File System Services (64-Bit)

Defined In: zParams.h

Syntax

```
struct                          /* zGET_DATA_STREAM_INFO */
{
   LONG   count;
   LONG   totalNameSize;
   QUAD   totalDataSize;
 } dataStream;
```

Fields

count

Specifies the number of data streams are owned by the file object.

totalNameSize

Specifies the number of characters that are used by the names of all the data streams owned by the file object.

totalDataSize

Specifies the total number of bytes of data that are occupied by data streams owned by the file object.

deleted

Is used by zInfo_s (page 108) and contains information about the time stamp and user ID of a deleted file.

Service: File System Services (64-Bit)

Defined In: zParams.h

Syntax

```
struct                              /* zGET_DELETED_INFO */
{                                   /* zMOD_DELETED_INFO */
   QUAD        time;
   UserID_t    id;                  /* zMOD_DELETED_ID   */
} deleted;
```

Fields

time

Specifies the time that the file was deleted.

id

Specifies the user ID of the user that deleted the file. If this information is requested and the file is not a deleted file, both fields are filled with zeroes.

extAttr

Is used by zInfo_s (page 108) and contains information about the extended attributes for the file object.

Service: File System Services (64-Bit)

Defined In: zParams.h

Syntax

```
struct                              /* zGET_EXTENDED_ATTRIBUTE_INFO */
{
   LONG   count;
   LONG   totalNameSize;
   QUAD   totalDataSize;
} extAttr;
```

Fields

count

Specifies the number of extended attributes that are owned by the file object.

totalNameSize

Specifies the number of characters that are used in the names of all the extended attributes that are owned by the file object.

totalDataSize

Specifies the total number of bytes of data that are occupied by the extended attributes that are owned by the file object.

id

Is used by zInfo_s (page 108) and contains the IDs for various owners and modifiers of the file.

Service: File System Services (64-Bit)

Defined In: zParams.h

Syntax

```
struct                                  /* zGET_IDS */
{
   UserID_t   owner;                   /* zMOD_OWNER_ID */
   UserID_t   archiver;                /* zMOD_ARCHIVER_ID */
   UserID_t   modifier;                /* zMOD_MODIFIER_ID */
   UserID_t   metaDataModifier;
                                  /* zMOD_METADATA_MODIFIER_ID */
} id;
```

Fields

owner

Specifies that the owner ID is set.

archiver

Specifies that the archiver ID is set.

modifier

Specifies that the modifier ID is set.

metaDataModifier

Specifies that the metaDataModifier is set.

names

Is used by zInfo_s (page 108) and contains information about the number of names in the variableData area.

Service: File System Services (64-Bit)

Defined In: zParams.h

Syntax

```
struct                          /* zGET_ALL_NAMES */
{
   LONG    numEntries;
   LONG    fileNameArray;
} names;
```

Fields

numEntries

Specifies the number of array entries that are valid in the fileNameArray.

fileNameArray

Specifies an index to an array of indexes that are each an index into the variableData section that references a Unicode string. This array is allocated in the variableData section and is indexed by the namespace ID. If the file object has a name that is valid in a given namespace, the array slot corresponding to that namespace ID is filled in with an index to the Unicode string for that name. If the file object does not have a valid name in a given namespace, the array slot is filled in with zero.

For example, if numEntries is set to 5, then slots 0, 1, 2, 3, and 4 of the fileNameArray are initialized to either contain zero or an index to a Unicode string. The name is also stored in the variableData section. If the same identical name is shared by more than one namespace, it is stored in the variableData area only once, but multiple array elements are set to index to the shared string, one for each namespace that is sharing the name.

For example, if info is a pointer to a zInfo_s structure, the following syntax accesses the DOS namespace filename:

```
(unicode_t *)&(((BYTE *)info)[((LONG *)&((BYTE *)info)
[info->names.fileNameArray])[zNSPACE_DOS]]
```

std

Is used by zInfo_s (page 108) and contains standard information about the file.

Service: File System Services (64-Bit)

Defined In: zParams.h

Syntax

```
struct
{
   Zid_t        zid;
   Zid_t        dataStreamZid;
   Zid_t        parentZid;
   QUAD         logicalEOF;
   VolumeID_t   volumeID;
   LONG         fileType;
   LONG         fileAttributes;
                             /* zMOD_FILE_ATTRIBUTES */
   LONG         fileAttributesModMask;
                             /* zMOD_FILE_ATTRIBUTES */
   LONG         padding;
} std;
```

Fields

zid

Specifies the ZID that uniquely identifies this data stream object within the volume on which it resides. Ignored by zModifyInfo.

dataStreamZid

Specifies the ZID of the data stream.

parentZid

Specifies the ZID of the parent that was used to open the file.

logicalEOF

Specifies the logical size of the file object's selected data stream in bytes, which is not necessarily the amount of physical storage occupied by the data stream. This size is the logical end of file for the data stream. Ignored by zModifyInfo.

volumeID

Specifies the GUID that identifies the volume where the file object resides. Ignored by zModifyInfo.

fileType

Specifies the file type of this file object. Ignored by zModifyInfo.

fileAttributes

Specifies the attributes that are currently associated with this file object. This field is modified by setting the zMOD_FILE_ATTRIBUTES bit in the modifyInfoMask for zModifyInfo.

fileAttributesModMask

Specifies a bit mask that defines which bits in the fileAttributes are to be set and/or cleared if the zMOD_FILE_ATTRIBUTES bit is set in the modifyInfoMask. If a bit in the fileAttributesModMask is set, the value of the corresponding bit (zero or 1) in fileAttributes is modified for the file object.

padding

Specifies some empty space.

storageUsed

Is used by zInfo_s (page 108) and contains storage information.

Service: File System Services (64-Bit)

Defined In: zParams.h

Syntax

```
struct                    /* zGET_STORAGE_USED */
{
   QUAD   physicalEOF;
   QUAD   dataBytes;
   QUAD   metaDataBytes;
} storageUsed;
```

Fields

physicalEOF

Specifies the byte offset of the physical end of file. ZLSS allows for data blocks to be allocated beyond the actual (logical) end of file, which permits the user to preallocate space for the file before it is actually needed. This process simplifies error handling and can give a hint to the system to allocate larger chunks of continuous memory.

dataBytes

Specifies the actual physical storage occupied by the file's data.

metaDataBytes

Specifies the actual physical storage used by the file's metadata. Ignored by zModifyInfo.

time

Is used by zInfo_s (page 108) and contains information about various time stamps for the file.

Service: File System Services (64-Bit)

Defined In: zParams.h

Syntax

```
struct
{
   QUAD   created;                 /* zMOD_CREATED_TIME */
   QUAD   archived;                /* zMOD_ARCHIVED_TIME */
   QUAD   modified;                /* zMOD_MODIFIED_TIME */
   QUAD   accessed;                /* zMOD_ACCESSED_TIME */
   QUAD    metaDataModified;
                           /*zMOD_METADATA_MODIFIED_TIME */
} time;
```

Fields

created

Specifies that the created time is set.

archived

Specifies that the archived time is set.

modified

Specifies that the modified time is set.

accessed

Specifies that the accessed time is set.

metaDataModified

Specifies that the metadata time is set.

zFILEMAP_ALLOCATION

Contains a map of what portions of the file are physically allocated versus what portions are sparse.

Service: File System Services (64-Bit)

Defined In: zPublics.h

Syntax

```
typedef struct zAllocationExtentElement_s
{
   QUAD   byteOffset;
   QUAD   lengthBytes;
} zAllocationExtentElement_s;
```

Fields

byteOffset

Specifies the starting byte offset in the file where this extent begins.

lengthBytes

Specifies the length (in bytes) of the extent.

Remarks

Each extent element identifies a starting offset and a length for an allocated portion of the file object. Every contiguously allocated portion of the file object is represented by exactly one extent list element. Unallocated or sparse portions of a file do not have an extent element returned for them.

zFILEMAP_LOGICAL

Contains a map of the logical extents being used on the volume and shows how fragmented a file is.

Service: File System Services (64-Bit)

Defined In: zPublics.h

Syntax

```
typedef struct zLogicalExtentElement_s
 {
    QUAD   blockNumber;
    QUAD   numBlocks;
} zLogicalExtentElement_s;
```

Fields

blockNumber

Specifies the logical block number in the volume where this extent starts. If this field contains FILEMAP_SPARSEBLOCK, this is an extent that describes an unallocated or sparse section of the file.

numBlocks

Specifies the number of logically contiguous blocks in this extent.

Remarks

Each extent element identifies a different fragment of the file on the logical volume. All fragments, including sparse ones, are represented by an extent.

zFILEMAP_PHYSICAL

Contains a map of the physical blocks that are being used, including what physical devices the blocks are on.

Service: File System Services (64-Bit)

Defined In: zPublics.h

Syntax

```
typedef struct zPhysicalExtent_s
{
   QUAD    length;
   QUAD    logicalOffset;
   QUAD    poolOffset;
   struct
   {
       QUAD    offset;
       LONG    deviceID;
       LONG    padding;
   } physical;
} zPhysicalExtent_s;
```

Fields

length

Specifies the length of the physical extent in bytes.

logicalOffset

Specifies the logical offset in the file in bytes.

poolOffset

Specifies the logical offset in the pool in bytes.

physical.offset

Specifies the physical offset in bytes, relative to the device identified by deviceID.

physical.deviceID

Specifies the device ID used by the Media Manager to identify the physical device where the extent resides.

zInfo_s

Contains detailed information about a file object.

Service: File System Services (64-Bit)

Defined In: zPublics.h

Syntax

```
#ifndef zGET_INFO_VARIABLE_DATA_SIZE
#define zGET_INFO_VARIABLE_DATA_SIZE (zMAX_COMPONENT_NAME*2)
#endif

typedef struct zInfo_s
{
   LONG    infoVersion;
   SLONG   totalBytes;
   SLONG   nextByte;
   LONG    padding;
   QUAD    retMask;
   struct  {           /* zGET_STD_INFO */
   } std;
   struct {            /* zGET_STORAGE_USED */
   } storageUsed;
   LONG   primaryNameSpaceID;    /* zGET_PRIMARY_NAMESPACE */
                                 /* zMOD_PRIMARY_NAMESPACE */
   LONG    nameStart;        /* zGET_NAME - index into zInfo */
   struct {            /* zGET_ALL_NAMES */
   } names;
   struct {
   } time;
   struct {           /* zGET_IDS */
   } id;
   struct {           /* zGET_BLOCK_SIZE */
   } blockSize;
   struct {           /* zGET_COUNTS */
   } count;
   struct {                    /* zGET_DATA_STREAM_INFO */
   } dataStream;
   struct {                 /* zGET_EXTENDED_ATTRIBUTE_INFO */
   } extAttr;
   struct {       /* zGET_DELETED_INFO, zMOD_DELETED_INFO & zMOD_DELETED_ID */
   } deleted;
   struct
   {
      struct zMacInfo_s  info;       /* zGET_MAC_METADATA */
   } macNS;
   struct                      /* zGET_UNIX_METADATA */
   {                           /* zMOD_UNIX_METADATA */
      struct   zUnixInfo_s info;
      LONG     offsetToData;
   } unixNS;

   zVolumeInfo_s   vol;      /* zGET_VOLUME_INFO */
   zPoolInfo_s     pool;     /* zGET_POOL_INFO */
   LONG   extAttrUserFlags;   /* zGET_EXTATTR_FLAGS */
                              /* zMOD_EXTATTR_FLAGS */
   BYTE   variableData [zGET_INFO_VARIABLE_DATA_SIZE];
} zInfo_s;
```

Fields

infoVersion

Specifies the version of the structure. Because this structure will need to change in the future, the version field identifies the particular structure being used.

totalBytes

Specifies the total number of bytes in fixed and variable data sections. Both sections are used so that the fixed area can grow without breaking older code pieces.

nextByte

Specifies the next byte to use in the variable area.

padding

Specifies some empty space.

retMask

Returns a bit mask of the fields actually returned by zGetInfo or zWildRead. Ignored by zModifyInfo.

std

Specifies the standard set of information. This field is selectable. With zWildRead, only the name of the file can be returned.

storageUsed

Returns three fields, if zGET_STORAGE_USED is set.

primaryNameSpaceID

Specifies the namespace used when the file was created. Names in the other namespaces are derived from the version of the file name in the primary namespace. On zModifyInfo, if zMOD_PRIMARY_NAMESPACE is set, the primary namespace is changed to the new value.

nameStart

Specifies an index with zGetInfo if zGET_NAME is set.

names

Returns two fields with zGetInfo if zGET_ALL_NAMES is set. Ignored by zModifyInfo.

time

Returns five fields: created, archived, modified, accessed, and metaDataModified with zGetInfo if zGET_TIMES is set. These are the UTC times when the file object was first created, last archived, last modified, last accessed, and the last time the metadata was modified. On zModifyInfo, each of the five fields can be modified separately, with a modify bit for each field.

id

Returns four fields: owner, archiver, modifier, and metaDataModifier with zGetInfo if zGET_IDS is specified. These are the eDirectory™ object IDs of the file object's current owner, last archiver, last modifier, and last metadata modifier. With zModifyInfo, each of the four fields can be modified separately, with a modify bit for each field.

blockSize

Returns size and sizeShift with zGetInfo if zGET_BLOCK_SIZE is set. Ignored by zModifyInfo.

count

Returns open and hardLink with zGetInfo if zGET_COUNTS is set. Ignored by zModifyInfo.

dataStream

Returns three fields with zGetInfo if zGET_DATA_STREAM_INFO is set.

extAttr

Returns three fields with zGetInfo if zGET_EXTENDED_ATTRIBUTE_INFO is set. Ignored by zModifyInfo.

deleted

Returns the time and ID with zGetInfo if zGET_DELETED_INFO is set. On zModifyInfo, if zMOD_DELETED_INFO is specified, and if the file is a deleted file, the deleted info is modified.

macNS

Returns four fields: finderInfo, proDOSInfo, filler, and dirRightsMask with zGetInfo if zGET_MAC_METADATA is set. The filler field is filled in with zeroes because its only purpose is to LONG-align the rest of the structure. If the zNSPACE_MAC namespace is not enabled on the volume, this structure is filled in with zeroes. On zModifyInfo, if zMOD_MAC_METADATA is specified, all fields (except filler) are modified in the metadata. If the zNSPACE_MAC namespace is not enabled on the volume, this structure is ignored.

unixNS

Returns the following fields: fMode, rDev, myFlags, nfsUID, nfsGID, nwUID, nwGID, nwEveryone, nwUIDRights, nwGIDRights, nwEveryoneRights, acsFlags, firstCreated, and variableSize with zGetInfo if zGET_UNIX_METADATA is set. If the zNSPACE_UNIX namespace is not enabled on the volume, this structure is filled in with zeroes. If variableSize is nonzero, variableStart indexes additional variable-sized Unix metadata that is stored in the variableData section of this structure.

On zModifyInfo, if zMOD_UNIX_METADATA is specified, all fields are modified in the metadata, including the variableStart (if variableSize is nonzero and the variableStart index is nonzero). If the zNSPACE_UNIX namespace is not enabled on the volume, this structure is ignored.

vol

Specifies the information that is returned is for the volume where the file object resides. However, if the file object is a volume object in the admin volume, the volume information for the volume object is returned. If zGET_VOLUME_INFO is set, the following data can be returned:

- If zMOD_VOL_MIN_KEEP_SECONDS is set in modifyInfoMask, minKeepSeconds is modified.
- If zMOD_VOL_MAX_KEEP_SECONDS is set, maxKeepSeconds is modified.
- If zMOD_VOL_LOW_WATER_MARK is set, lowWaterMark is modified.
- If zMOD_VOL_HIGH_WATER_MARK is set, highWaterMark is modified.

extAttrUserFlags

Specifies an arbitrary value that is set by the user. This field only applies to extend attributes and has no particular significance to the file system.

variableData

Specifies additional data space, which is allocated by the caller, to be used for all returned variable length optional data for zGetInfo. It is the responsibility of the caller to make sure this variable section is large enough to hold all of the requested variable data. If it is not large enough

an error is returned, and only the data that fits in this area is filled in. The caller references this area from indexes set in the fixed area. The indexes are relative to the front of the zInfo structure, not from the beginning of the variable data area.

Remarks

All fields that are modifiable using zModifyInfo are marked with a /* zMOD_xxx */ comment tag, and all of the optional fields that can be retrieved using zGetInfo are marked with a /* zGET_xxx */ comment tag. When an entire substructure is modifiable or retrievable using a single bit, the comment tag is on the structure itself.

In order for it to be expanded in the future, this structure is identified by a version number as defined by zINFO_VERSION_A.

zInfo_s must be preallocated by the caller. It contains a fixed portion for fixed size data and a variable portion to contain variable size data indexed to by the fixed portion.

Data values fall on their natural boundaries without introducing fields for padding, which usually means the largest data items come first in the structure. Because this structure can be passed between 32-bit and 64-bit environments, all fields must be of the specified sizes and aligned correctly.

Variable length data items are added to the end with indexes to the corresponding offsets stored in the body of the structure. An index value of zero indicates there is no variable data for the item.

zGetInfo and zWildRead

The following rules apply to this structure when zGetInfo and zWildRead are called:

- The fixed portion of this structure is filled based on the bits set in the getInfoMask parameter.
- The variable portion must be large enough to store any variable size optional information requested. Data is not directly referenced in the variable portion; instead, it is referenced using indexes that are stored in the fixed portion. To make it easier to grow the fixed portion of the zInfo structure, the indexes starts from the beginning of the data structure rather than from the variable data array.

  For example, if zGET_NAME was specified, the name information is stored in the variable portion, and the name index indexes the location in the variable section where the name was stored. This methodology allows the NSS system to pack variable size data into the variable data and still gives you easy access to all of the data through indexes.
- When you are declaring stack variables and using the sizeof operator, the default size of the variable portion is equal to the value of the define zGET_INFO_VARIABLE_DATA_SIZE (zMAX_COMPONENT_NAME*2 bytes). If this size is inappropriate, it can be overridden by defining it before including zPublics.h as follows:

  ```
  #undef  zGET_INFO_VARIABLE_DATA_SIZE
  #define zGET_INFO_VARIABLE_DATA_SIZE desired size
  ```
- If the variable section is not large enough to contain all of the requested data, an error is returned that indicates some but not all of the requested data was returned. If this happens, portions of the variable data for which indexes were filled in can be considered valid, and the indexes to data that was not returned are set to zero.

zModifyInfo

The following rules apply to this structure when zModifyInfo is called:

- The structure format is the same as that on zGetInfo, but zModifyInfo modifies only the fields identified by the bits set in the modifyInfoMask parameter. All other structure data that is not specified in the modifyInfoMask is ignored.
- The variable portion is ignored.

zInfoB_s

Contains detailed information about a file object and allows directory quota information to be returned.

Service: File System Services (64-Bit)

Defined In: zPublics.h

Syntax

```
#ifndef zGET_INFO_VARIABLE_DATA_SIZE
#define zGET_INFO_VARIABLE_DATA_SIZE (zMAX_COMPONENT_NAME*2)
#endif

typedef struct zInfo_s
{
   LONG    infoVersion;
   SLONG   totalBytes;
   SLONG   nextByte;
   LONG    padding;
   QUAD    retMask;
   struct  {            /* zGET_STD_INFO */
   } std;
   struct {             /* zGET_STORAGE_USED */
   } storageUsed;
   LONG   primaryNameSpaceID;   /* zGET_PRIMARY_NAMESPACE */
                                   /* zMOD_PRIMARY_NAMESPACE */
   LONG    nameStart;       /* zGET_NAME - index into zInfo */
   struct {             /* zGET_ALL_NAMES */
   } names;
   struct {
   } time;
   struct {            /* zGET_IDS */
   } id;
   struct {            /* zGET_BLOCK_SIZE */
   } blockSize;
   struct {            /* zGET_COUNTS */
   } count;
   struct {                      /* zGET_DATA_STREAM_INFO */
   } dataStream;
   struct {                   /* zGET_EXTENDED_ATTRIBUTE_INFO */
  } extAttr;
   struct {       /* zGET_DELETED_INFO, zMOD_DELETED_INFO & zMOD_DELETED_ID*/
   } deleted;
   struct
   {
      struct zMacInfo_s           /* zGET_MAC_METADATA */
   } macNS;
   struct                        /* zGET_UNIX_METADATA */
   {                             /* zMOD_UNIX_METADATA */
      struct   zUnixInfo_s
      LONG     offsetToData;
   } unixNS;
   zVolumeInfo_s    vol;      /* zGET_VOLUME_INFO */
   zPoolInfo_s      pool;     /* zGET_POOL_INFO */
   LONG   extAttrUserFlags;       /* zGET_EXTATTR_FLAGS */
                                  /* zMOD_EXTATTR_FLAGS */

   struct
   {
      SQUAD   quota;
      SQUAD   usedAmount;
   } dirQuota;
   BYTE   variableData [zGET_INFO_VARIABLE_DATA_SIZE];
} zInfo_s;
```

Fields

infoVersion

Specifies the version of the structure. Because this structure will need to change in the future, the version field identifies the particular structure being used.

totalBytes

Specifies the total number of bytes in fixed and variable data sections. Both sections are used so that the fixed area can grow without breaking older code pieces.

nextByte

Specifies the next byte to use in the variable area.

padding

Specifies some empty space.

retMask

Returns a bit mask of the fields actually returned by zGetInfo or zWildRead. Ignored by zModifyInfo.

std

Specifies the standard set of information. This field is selectable. With zWildRead, only the name of the file can be returned.

storageUsed

Returns three fields, if zGET_STORAGE_USED is set.

primaryNameSpaceID

Specifies the namespace used when the file was created. Names in the other namespaces are derived from the version of the filename in the primary namespace. On zModifyInfo, if zMOD_PRIMARY_NAMESPACE is set, the primary namespace is changed to the new value.

nameStart

Specifies an index with zGetInfo if zGET_NAME is set.

names

Returns two fields with zGetInfo if zGET_ALL_NAMES is set. Ignored by zModifyInfo.

time

Returns five fields: created, archived, modified, accessed, and metaDataModified with zGetInfo if zGET_TIMES is set. These are the UTC times when the file object was first created, last archived, last modified, last accessed, and the last time the metadata was modified. On zModifyInfo, each of the five fields can be modified separately, with a modify bit for each field.

id

Returns four fields: owner, archiver, modifier, and metaDataModifier with zGetInfo if zGET_IDS is specified. These are the NDS object IDs of the file object's current owner, last archiver, last modifier, and last metadata modifier. With zModifyInfo, each of the four fields can be modified separately, with a modify bit for each field.

blockSize

Returns size and sizeShift with zGetInfo if zGET_BLOCK_SIZE is set. Ignored by zModifyInfo.

count

Returns open and hardLink with zGetInfo if zGET_COUNTS is set. Ignored by zModifyInfo.

dataStream

Returns three fields with zGetInfo if zGET_DATA_STREAM_INFO is set.

extAttr

Returns three fields with zGetInfo if zGET_EXTENDED_ATTRIBUTE_INFO is set. Ignored by zModifyInfo.

deleted

Returns the time and ID with zGetInfo if zGET_DELETED_INFO is set. On zModifyInfo, if zMOD_DELETED_INFO is specified, and if the file is a deleted file, the deleted info is modified.

macNS

Returns four fields: finderInfo, proDOSInfo, filler, and dirRightsMask with zGetInfo if zGET_MAC_METADATA is set. The filler field is filled in with zeroes because its only purpose is to LONG-align the rest of the structure. If the zNSPACE_MAC namespace is not enabled on the volume, this structure is filled in with zeros. On zModifyInfo, if zMOD_MAC_METADATA is specified, all fields (except filler) are modified in the metadata. If the zNSPACE_MAC namespace is not enabled on the volume, this structure is ignored.

unixNS

Returns the following fields: fMode, rDev, myFlags, nfsUID, nfsGID, nwUID, nwGID, nwEveryone, nwUIDRights, nwGIDRights, nwEveryoneRights, acsFlags, firstCreated, and variableSize with zGetInfo if zGET_UNIX_METADATA is set. If the zNSPACE_UNIX namespace is not enabled on the volume, this structure is filled in with zeroes. If variableSize is nonzero, variableStart indexes additional variable-sized UNIX metadata that is stored in the variableData section of this structure.

On zModifyInfo, if zMOD_UNIX_METADATA is specified, all fields are modified in the metadata, including the variableStart (if variableSize is nonzero and the variableStart index is nonzero). If the zNSPACE_UNIX namespace is not enabled on the volume, this structure is ignored.

vol

Specifies the information that is returned is for the volume where the file object resides. However, if the file object is a volume object in the admin volume, the volume information for the volume object is returned. If zGET_VOLUME_INFO is set, the following data can be returned:

- If zMOD_VOL_MIN_KEEP_SECONDS is set in modifyInfoMask, minKeepSeconds is modified.
- If zMOD_VOL_MAX_KEEP_SECONDS is set, maxKeepSeconds is modified.
- If zMOD_VOL_LOW_WATER_MARK is set, lowWaterMark is modified.
- If zMOD_VOL_HIGH_WATER_MARK is set, highWaterMark is modified.

extAttrUserFlags

Specifies an arbitrary value that is set by the user. This field only applies to extend attributes and has no particular significance to the file system.

dirQuota

Specifies quota information for a directory. Returns two fields: quota and usedAmount. The quota field contains the directory quota. The usedAmount field contains the amount of storage used within the directory.

variableData

Specifies additional data space, which is allocated by the caller, to be used for all returned variable length optional data for zGetInfo. It is the responsibility of the caller to make sure this variable section is large enough to hold all of the requested variable data. If it is not large enough, an error is returned and only the data that fits in this area is filled in. The caller references this area from indexes set in the fixed area. The indexes are relative to the front of the zInfo structure, not from the beginning of the of the beginning of the variable data area.

Remarks

All fields that are modifiable using zModifyInfo are marked with a /* zMOD_xxx */ comment tag, and all of the optional fields that can be retrieved using zGetInfo are marked with a /* zGET_xxx */ comment tag. When an entire substructure is modifiable or retrievable using a single bit, the comment tag is on the structure itself.

In order for it to be expanded in the future, this structure is identified by a version number as defined by zINFO_VERSION_A.

zInfo_s must be preallocated by the caller. It contains a fixed portion for fixed size data and a variable portion to contain variable size data indexed to by the fixed portion.

Data values fall on their natural boundaries without introducing fields for padding, which usually means the largest data items come first in the structure. Because this structure can be passed between 32-bit and 64-bit environments, all fields must be of the specified sizes and aligned correctly.

Variable length data items are added to the end with indexes to the corresponding offsets stored in the body of the structure. An index value of zero indicates there is no variable data for the item.

zGetInfo and zWildRead

The following rules apply to this structure when zGetInfo and zWildRead are called:

- The fixed portion of this structure is filled based on the bits set in the getInfoMask parameter.
- The variable portion must be large enough to store any variable size optional information requested. Data is not directly referenced in the variable portion; instead, it is referenced using indexes that are stored in the fixed portion. To make it easier to grow the fixed portion of the zInfo structure, the indexes start from the beginning of the data structure rather than from the variable data array.

  For example, if zGET_NAME was specified, the name information is stored in the variable portion, and the name index indexes the location in the variable section where the name was stored. This methodology allows the NSS system to pack variable size data into the variable data and still gives you easy access to all of the data through indexes.
- When you are declaring stack variables and using the sizeof operator, the default size of the variable portion is equal to the value of the define zGET_INFO_VARIABLE_DATA_SIZE (zMAX_COMPONENT_NAME*2 bytes). If this size is inappropriate, it can be overridden by defining it before including zPublics.h as follows:

  ```
  #undef  zGET_INFO_VARIABLE_DATA_SIZE
  #define zGET_INFO_VARIABLE_DATA_SIZE desired size
  ```
- If the variable section is not large enough to contain all of the requested data, an error is returned that indicates some but not all of the requested data was returned. If this happens, portions of the variable data for which indexes were filled in can be considered valid, and the indexes to data that was not returned are set to zero.

zModifyInfo

The following rules apply to this structure when zModifyInfo is called:

- The structure format is the same as that on zGetInfo, but zModifyInfo modifies only the fields identified by the bits set in the modifyInfoMask parameter. All other structure data that is not specified in the modifyInfoMask is ignored.
- The variable portion is ignored.

zMacInfo_s

Contains information about Macintosh* files.

Service: File System Services (64-Bit)

Defined In: zParam.h

Syntax

```
typedef struct zMacInfo_s
{
   BYTE   finderInfo[32];
   BYTE   proDOSInfo[6];
   BYTE   filler[2];
   LONG   dirRightsMask;
} zPhysicalExtent_s;
```

Fields

finderInfo

Specifies the Macintosh FInfo data (as stored and retrieved for Macintosh files).

proDOSInfo

Specifies the Macintosh proDOSInfo as a 2-byte file type and 4-byte aux type for Pro DOS workstations. Pro DOS workstations are now mostly obsolete.

filler

Is unused.

dirRightsMask

Is unused. It is set and stored for the Macintosh client for backward compatibility. This is not used for NSS files or AFPTCP. The Pro Soft client might not use this field either.

zPoolInfo_s

Contains information about a pool.

Syntax

```
typedef struct zPoolInfo_s
{
   VolumeID_t   poolID;
   NDSid_t      ndsObjectID;
   LONG         poolState;
   LONG         nameSpaceMask;
   struct{
      QUAD   enabled;
      QUAD   enableModMask;
      QUAD   supported;
   }features;
   QUAD   totalSpace;
   QUAD   numUsedBytes;
   QUAD   purgeableBytes;
   QUAD   nonPurgeableBytes;
}zPoolInfo_s;
```

Fields

poolID

Specifies a 128-bit GUID that identifies the pool.

ndsObjectID

Specifies a 128-bit object ID that is assigned by eDirectory (zMOD_POOL_NDS_OBJECT_ID).

poolState

Specifies the current state of the pool:

- DEACTIVE (2)
- MAINTENANCE (3)
- ACTIVE (6)

nameSpaceMask

Specifies the namespaces that are supported on the pool.

enabled

Specifies the features that are currently enabled for the pool (zMOD_POOL_ATTRIBUTES).

enableModMask

Specifies the features that should be changed (zMOD_POOL_ATTRIBUTES).

supported

Specifies the features that can be supported by the pool.

totalSpace

Specifies the total number of bytes that are available on the pool.

numUsedBytes

Specifies the number of bytes that are used in the pool.

purgeableBytes

Specifies the number of bytes in files that are deleted but not yet purged.

nonPurgeableBytes

Specifies the number of bytes in deleted files that cannot yet be purged.

zUnixInfo_s

Contains information about the UNIX system.

Service: File System Services (64-Bit)

Defined In: zParam.h

Syntax

```
typedef struct zUnixInfo_s
{
   LONG    fMode;
   LONG    rDev;
   LONG    myFlags;
   LONG    nfsUID;
   LONG    nfsGID;
   LONG    nwUID;
   LONG    nwGID;
   LONG    nwEveryone;
   LONG    nwUIDRights;
   LONG    nwGIDRights;
   LONG    nwEveryoneRights;
   BYTE    acsFlags;
   BYTE    firstCreated;
   SWORD   variableSize;
} zPhysicalExtent_s;
```

zVolumeInfo_s

Contains information about volumes.

Syntax

```
typedef struct zVolumeInfo_s
{
   VolumeID_t   volumeID;
   NDSid_t      ndsObjectID;
   LONG         volumeState;
   LONG         nameSpaceMask;
   struct{
      QUAD   enabled;
      QUAD   enableModMask;
      QUAD   supported;
   }features;
   QUAD   maximumFileSize;
   QUAD   totalSpaceQuota;
   QUAD   numUsedBytes;
   QUAD   numObjects;
   QUAD   numFiles;
   LONG   authModelID;
   LONG   dataShreddingCount;
   struct{
      QUAD   purgeableBytes;
      QUAD   nonPurgeableBytes;
      QUAD   numDeletedFiles;
      QUAD   oldestDeletedTime;
      LONG   minKeepSeconds;
      LONG   maxKeepSeconds;
      LONG   lowWaterMark;
      LONG   highWaterMark;
   }salvage;
   struct{
      QUAD   numCompressedFiles;
      QUAD   numCompDelFiles;
      QUAD   numUncompressableFiles;
      QUAD   numPreCompressedBytes;
      QUAD   numCompressedBytes;
   }comp;
}zVolumeInfo_s;
```

Fields

volumeID

Specifies a 128-bit GUID that identifies the volume.

ndsObjectID

Specifies a 128-bit object ID that is assigned by eDirectory (zMOD_VOL_NDS_OBJECT_ID).

volumeState

Specifies the current state of the volume:

- DEACTIVE (2)
- MAINTENANCE (3)
- ACTIVE (6)

nameSpaceMask

Specifies the namespace that is supported on the volume.

enabled

Specifies the features that are currently enabled for the volume (zMOD_VOL_ATTRIBUTES).

enableModMask

Specifies the features that should be changed (zMOD_VOL_ATTRIBUTES).

supported

Specifies the features that can be supported on the volume.

maximumFileSize

Specifies the maximum file size that is supported on the volume.

totalSpaceQuota

Specifies the maximum size of the volume, which can still overbook the pool (zMOD_VOL_QUOTA).

numUsedBytes

Specifies the number of bytes that are in use by the volume.

numObjects

Specifies the number of file-like objects on the volume, including extended attributes and data streams.

numFiles

Specifies the number of files on the volume.

authModelID

Specifies the authorization:

- zFTYPE_ZAS_AUTH_MODEL is for traditional NetWare®
- zFTYPE_UNIX_AUTH_MODEL is for UNIX (rwxrwxrwx)

dataShreddingCount

Specifies how many writes are done to the disk to obscure the original date (zMOD_VOL_DATA_SHREDDING_COUNT). Up to seven passes can be specified.

purgeableBytes

Specifies the space that is used by deleted files that can be reclaimed.

nonPurgeableBytes

Specifies the space that is used by deleted files that cannot be reclaimed yet.

numDeletedFiles

Specifies the total number of deleted files that are in salvage.

oldestDeletedTime

Specifies the date of the oldest deleted file in salvage.

minKeepSeconds

Specifies the minimum number of seconds that a file can be kept in the salvage system (zMOD_VOL_MIN_KEEP_SECONDS). This is not supported on ZLSS.

maxKeepSeconds

Specifies the maximum amount of time to keep deleted files (zMOD_VOL_MAX_KEEP_SECONDS).

lowWaterMark

Specifies an amount of used space under which deleted files should be kept (zMOD_VOL_LOW_WATER_MARK). When the percent of used space reaches the highWaterMark, NSS begins purging deleted files until the amount of used space drops below the lowWaterMark.

highWaterMark

Specifies the maximum amount of used space that deleted files should occupy (zMOD_VOL_HIGH_WATER_MARK). When the percent of used space reaches the highWaterMark, NSS begins purging deleted files until the amount of used space drops below the lowWaterMark.

numCompressedFiles

Specifies the number of compressed files.

numCompDelFiles

Specifies the number of compressed deleted files.

numUncompressibleFiles

Specifies the number of files that cannot be compressed.

numPreCompressedBytes

Specifies the amount of space that would have been used if no files were compressed.

numCompressedBytes

Specifies the amount of space that is used with compressed files.

Remarks

The salvage system saves deleted files until the system needs their space. The salvage structure provides information about those deleted files (zGET_VOL_SALVAGE_INFO).

4 Values

This documentation describes the following values associated with File System Services:

- Section 4.1, “Return Values,” on page 126
- Section 4.2, “Common Values,” on page 146
- Section 4.3, “Data Types,” on page 157

4.1 Return Values

All functions return STATUS code (defined as an SNINT). This documentation lists the following possible STATUS error codes and describes what each indicates:

- Section 4.1.1, “zErrors (0-20013),” on page 127
- Section 4.1.2, “Message, User Transaction, and Task Errors (20051-20070),” on page 128
- Section 4.1.3, “eDirectory Errors (20090-20099),” on page 128
- Section 4.1.4, “General File System Errors (20100-20105),” on page 128
- Section 4.1.5, “Virtual File System Errors (20150-20159),” on page 129
- Section 4.1.6, “General Storage System Errors (20200-20213),” on page 129
- Section 4.1.7, “Admin Volume Errors (20250-20252),” on page 130
- Section 4.1.8, “Beast Errors (20300-20312),” on page 130
- Section 4.1.9, “Naming Errors (20400-20446),” on page 130
- Section 4.1.10, “Name Typing Errors (20499),” on page 132
- Section 4.1.11, “Rename Errors (20500-20512),” on page 132
- Section 4.1.12, “Data Stream Errors (20550-20551),” on page 133
- Section 4.1.13, “Semantic Agent Errors (20601-20602),” on page 133
- Section 4.1.14, “Direct FS I/O & DIO Errors (20650-20655),” on page 133
- Section 4.1.15, “Namespace Errors (20700-20705),” on page 133
- Section 4.1.16, “Asynchronous Errors (20750-20751),” on page 134
- Section 4.1.17, “Encrypted Volume Errors (20798-20799),” on page 134
- Section 4.1.18, “Volume and Pool Errors (20800-20839),” on page 134
- Section 4.1.19, “DSI and Adding Volumes Errors (20840-20849),” on page 136
- Section 4.1.20, “Authorization Errors (20850-20871),” on page 136
- Section 4.1.21, “NWCS Errors (20890-20898),” on page 137
- Section 4.1.22, “Locking Errors (20900-20911),” on page 137
- Section 4.1.23, “Unicode Errors (20950-20955),” on page 138
- Section 4.1.24, “Link Errors (21000-21005),” on page 138
- Section 4.1.25, “NLM Registration Errors (21050-21058),” on page 138
- Section 4.1.26, “MASV Errors (21100),” on page 139
- Section 4.1.27, “Modify Volume Info Errors (21150-21151),” on page 139
- Section 4.1.28, “Feature Not Enabled Errors (21200-21215),” on page 139
- Section 4.1.29, “User Space Restriction Errors (21300-21303),” on page 140
- Section 4.1.30, “User Store Errors (21350-21351),” on page 140
- Section 4.1.31, “Compression Manager Generated Errors (21400-21410),” on page 140
- Section 4.1.32, “Directory Quota Errors (21500-21505),” on page 141
- Section 4.1.33, “User Transaction Errors (21600-21606),” on page 141
- Section 4.1.34, “Management File Errors (21700-21704),” on page 141
- Section 4.1.35, “Data Migration Errors (21800-21803),” on page 142
- Section 4.1.36, “Semantic Agent Errors (22000),” on page 142

- Section 4.1.37, “CD-Specific Errors (22500-22512),” on page 142
- Section 4.1.38, “DOSFAT/FAT32-Specific Errors (22600-22607),” on page 142
- Section 4.1.39, “NSS Java Interface Errors (22900-22999),” on page 143
- Section 4.1.40, “SMS Tape LSS Errors (23000-23024),” on page 143
- Section 4.1.41, “NFS Gateway LSS Errors (23100-23199),” on page 143
- Section 4.1.42, “Storage System B-Tree Errors (24800-24803),” on page 143
- Section 4.1.43, “Storage System ZLOG Errors (24820-24831),” on page 143
- Section 4.1.44, “ZVL Errors (24838-24839),” on page 144
- Section 4.1.45, “ZFSVOL/ZLSSPOOL Volume Data Errors (24840-24849),” on page 144
- Section 4.1.46, “Checkpoint Errors (24850-24853),” on page 145
- Section 4.1.47, “Superblock Errors (24860-24868),” on page 145
- Section 4.1.48, “Recovery Errors (24880),” on page 145
- Section 4.1.49, “FSHooks Errors (24999),” on page 145

4.1.1 zErrors (0-20013)

0	zOK: The operation succeeded.
20000	zERR_NO_MEMORY: The operation failed because it was unable to allocate additional memory.
20001	zERR_BAD_CONNECTION_ID: The connectionID parameter passed in to the function was bad or invalid.
20002	zERR_NOT_CONNECTED: The connection identified by the connectionID parameter is not currently active.
20003	zERR_XID_NOT_SUPPORTED: This function does not currently support transactions, and a transaction ID (XID) was specified. The value zNILXID must be specified for the xid parameter.
20004	zERR_BUFFER_TOO_SMALL: The buffer passed to the function is not big enough to contain all of the requested information. Depending on the semantics of the function, some partial results might have been returned in the buffer, which is too small.
20005	zERR_RETURN_PARA_NULL: A function’s return parameter, which should contain a pointer to some data to be filled in, contains NULL.
20006	zERR_QUAD_TOO_BIG_FOR_LONG: In converting a QUAD to a LONG, the value is too large to fit in the LONG.
20007	zERR_CONNECTION_NOT_LOGGED_IN: The specified connection might not be used because it is not authenticated and logged in.
20008	zERR_BAD_PARAMETER_VALUE: One of the parameters has an unsupported value.
20009	zERR_INVALID_SEMANTIC_AGENT_ID: An invalid semantic agent ID was specified.
20010	zERR_INVALID_STATE: An invalid state was requested.
20011	zERR_NOT_SUPPORTED: The operation is not supported.
20012	zERR_MEDIA_CORRUPTED: The media is corrupted.
20015	zERR_TIMEOUT: An event did not occur before a timer expired.

20016 zERR_EXCEEDED_MAX_ALERTS: Exceeded the maximum number of outstanding alerts.

20017 zERR_USER_ABORTED: User requested action to stop.

20018 zERR_BAD_ADDRESS: A bad user space (ring 3) address.

4.1.2 Message, User Transaction, and Task Errors (20051-20070)

20051 zERR_BAD_KEY: Key could not be found.

20052 zERR_BAD_METHOD: Method number out of range.

20053 zERR_BROKEN_DOOR: Key is still valid but the object does not exist anymore.

20054 zERR_NO_DUP: Cannot duplicate key.

20055 zERR_NO_METHOD: No method for this number.

20056 zERR_BROKEN_OBJECT: Object already broken.

20060 zERR_NO_TASK: Could not find task that was specified by user.

20061 zERR_TASK_EXISTS: This task already exists.

20062 zERR_NO_XACTION: User transaction ID does not exist.

20070 zERR_FINISHED_WITH_EXTENTS: Finished processing the number of extents that were passed in by the user.

4.1.3 eDirectory Errors (20090-20099)

20090 zERR_OBJECT_NOT_FOUND: Object not found in eDirectory™.

20091 zERR_UNABLE_TO_IMPORT_NDS_PUBLICS: Problem while importing eDirectory publics.

20092 zERR_UNABLE_TO_GET_LONG_NAME: Error getting the distinguished name from eDirectory.

20093 zERR_GUID_NOT_FOUND: Looking up by GUID did not succeed.

4.1.4 General File System Errors (20100-20105)

20100 zERR_END_OF_FILE: The operation has reached the end of the file.

20101 zERR_HARD_READ_ERROR: Read error from media.

20102 zERR_HARD_WRITE_ERROR: Write error from media.

20103 zERR_OUT_OF_SPACE: No available disk space is left.

20104 zERR_PURGED_SPACE_UNAVAILABLE: There is purgeable space but it has not yet been freed.

20105 zERR_FILE_TOO_LARGE: The file is too large for the given pool.

20106 zERR_INVALID_BLOCK: Requested a read on an invalid block.

20107 zERR_CONTIGUOUS_SPACE: Requested amount of contiguous blocks unavailable.

20108 zERR_ZID_GREATER_THAN_32BITS: Ran out of lower ZIDs.

4.1.5 Virtual File System Errors (20150-20159)

20150 zERR_BAD_TRANSFORMATION: Bad transformation in a virtual file.

20151 zERR_SYMBOL_NAME_TOO_LONG: The symbol name in a virtual file is too long.

20152 zERR_SYMBOL_NOT_DEFINED: The symbol for a virtual request is not defined.

20153 zERR_XML_TOO_LONG: A generated XML string is too large.

20154 zERR_DATASTREAM_NOT_FOUND: A searched-for data stream was found in a virtual file.

20155 zERR_BAD_FUNCTION_PTR: A virtual file has a bad function pointer.

20156 zERR_BAD_FORMAT: A virtual file has a bad format type.

20157 zERR_BAD_OFFSET: The passed in offset for a read or write is not valid.

20158 zERR_NO_FUNCTION_DEFINED: No function is present where one is needed.

20159 zERR_SYMBOL_NAME_MISSING: The symbol name in a virtual file is missing.

4.1.6 General Storage System Errors (20200-20213)

20200 zERR_VOLUME_ALREADY_INITIALIZED: Attempted to initialize a volume that is already set up.

20201 zERR_QUEUE_READ_FAILURE: Unable to queue a read request.

20202 zERR_QUEUE_WRITE_FAILURE: Unable to queue a write request.

20203 zERR_READ_FAILURE: The low level asynchronous block read failed.

20204 zERR_WRITE_FAILURE: The low level asynchronous block write failed.

20205 zERR_VOLUME_DISABLING: Volume is being disabled so that I/O is not allowed.

20206 zERR_POOL_DISABLING: The pool is being disabled so that I/O is not allowed.

20207 zERR_POOL_NOT_ACCESSIBLE: The pool is not accessible so that I/O is not allowed.

20208 zERR_READ_FAILURE_UNKNOWN: Unknown read error.

20209 zERR_READ_FAILURE_MEDIA: Media read error.

20210 zERR_READ_FAILURE_POSTPONE: Postpone read error.

20211 zERR_WRITE_FAILURE_UNKNOWN: Unknown write error.

20212 zERR_WRITE_FAILURE_MEDIA: Media write error.

20213 zERR_WRITE_FAILURE_POSTPONE: Postpone write error.

4.1.7 Admin Volume Errors (20250-20252)

20250 zERR_NO_ADMIN_VOLUME: No admin volume exists.

20251 zERR_NO_PERSIST_ADMIN_VOLUME: No persistent admin volume exists.

20252 zERR_UNABLE_TO_INIT_ADMIN_VOL: Could not start the NSS_ADMIN volume.

4.1.8 Beast Errors (20300-20312)

20300 zERR_INVALID_BEAST_ID: An invalid beast ID was specified.

20301 zERR_BEAST_CLASS_ALREADY_DEFINED: The given beast class ID is already defined.

20302 zERR_BEAST_CLASS_NOT_DEFINED: The given beast class ID is not defined.

20303 zERR_BEAST_CLASS_ROUTINE_NOT_DEF: A required beast class routine is not defined.

20304 zERR_BEAST_CLASS_ROUTINE_MULT_DEF: A beast class routine is defined multiple times.

20305 zERR_COMM_OP_NOT_SUPPORTED: The given COMM operation (beast or volume) is not supported.

20306 zERR_INHERITANCE_DEPTH_EXCEEDED: The maximum inheritance depth has been exceeded.

20307 zERR_BEAST_SIZE_CHANGED: The beast size has been illegally changed.

20308 zERR_BAD_LENGTH_UNPACKING_BEAST: The system detected an inconsistent length while unpacking a beast.

20309 zERR_UNSUPPORTED_OBJECT_LAYOUT: The object's layout on disk is in an unsupported format.

20310 zERR_BEAST_CORRUPTED: The beast is corrupted. This is the only error where the ZLSS rebuild deletes the beast. Rebuilds should repair the beast if possible. If the rebuild cannot do a repair, it can delete the beast.

20311 zERR_FILE_IN_USE: The beast has a use count greater than 0.

20312 zERR_BEAST_BEING_PURGED_IN_REBUILD: The beast was in a state of being purged. The rebuild deletes this beast without logging so that no information is available.

4.1.9 Naming Errors (20400-20446)

20400 zERR_INVALID_HANDLE_PATH: The specified handle path structure is invalid.

20401 zERR_BAD_FILE_HANDLE: The specified file handle is invalid.

20402 zERR_BAD_CONTEXT_HANDLE: The specified context handle is invalid.

20403 zERR_INVALID_NAME: The specified filename is syntactically invalid.

20404 zERR_INVALID_CHAR_IN_NAME: The name contains an invalid character.

20405 zERR_INVALID_PATH: The specified directory path is invalid. Possible causes are an invalid character or the name was too long.

20406	zERR_RESERVED_NAMED: The specified name is reserved in this namespace.
20407	zERR_NAME_NOT_FOUND_IN_DIRECTORY: The specified filename was not found in the directory.
20408	zERR_NOT_DIRECTORY_FILE: Attempted to perform a directory type operation on a file that is not a directory.
20409	zERR_NO_NAMES_IN_PATH: The input path does not contain any pat names.
20410	zERR_NO_MORE_NAMES_IN_PATH: There are no more names in the specified input path.
20411	zERR_PATH_MUST_BE_FULLY_QUALIFIED: The input path is required to be a fully qualified path.
20412	zERR_FILE_ALREADY_EXISTS: Attempted to add a file to a directory and the filename already exists in the directory.
20413	zERR_NAME_NO_LONGER_VALID: The specified filename might have been valid previously, but it is no longer valid.
20414	zERR_BAD_SEARCHMAP_ID: The specified search map ID was invalid.
20415	zERR_INVALID_TYPE_FILE: The file type is wrong for the requested operation.
20416	zERR_INVALID_FILE_TYPE: An invalid file type was specified.
20417	zERR_DIRECTORY_NOT_EMPTY: The directory cannot be deleted because it still has files in it.
20418	zERR_BAD_SEARCH_OPTIONS: An invalid search option was specified.
20419	zERR_INVALID_SEARCH_SEQ_NUM: An invalid search sequence number was given.
20420	zERR_INTERNAL_DIRECTORY_ERROR: An internal error has occurred when accessing a directory.
20421	zERR_INVALID_MODIFY_PARAMETER: An invalid parameter was passed to modify info.
20422	zERR_INVALID_USER_ID: The user ID is not valid.
20423	zERR_NOTHING_CHANGED: Modify info was called, but no information was changed by the call.
20424	zERR_NO_FILES_FOUND: No files matched the given wildcard pattern.
20425	zERR_UNABLE_TO_RETURN_INFO: The build info routines could not complete.
20426	zERR_FILE_DID_NOT_MATCH_ATTR: The specified file did not match the match attribute criteria.
20427	zERR_FILE_DID_NOT_MATCH_TYPE: The specified file did not match the input matchFileType matching criteria.
20428	zERR_FILE_DID_NOT_MATCH_TYPEATTR: The specified file did not match the match attribute criteria.
20429	zERR_LINK_IN_PATH: A link object was found as a component in a path.
20430	zERR_LINK_IN_DEST_PATH: A link object was found as a component in a destination path.
20431	zERR_UNABLE_TO_OPEN_BEAST: The specified beast could not be opened.
20432	zERR_NSPACE_NAME_ALREADY_DEFINED: The specified namespace name is already defined.
20433	zERR_NAME_NOT_FOUND_IN_BEAST: The requested name was not found in the beast.

20434	zERR_PARENT_NOT_FOUND_IN_BEAST: The requested parent was not found in the beast.
20435	zERR_DIR_CANNOT_BE_OPENED: The directory file cannot be opened.
20436	zERR_INVALID_CONTEXT_HANDLE_TYPE: The context handle type is invalid.
20437	zERR_CONTAINER_NOT_FILE_BEAST: The container beast is not derived from the file beast type.
20438	zERR_NO_OPEN_PRIVILEGE: The user does not have privileges to open the file.
20439	zERR_NO_MORE_CONTEXT_HANDLE_IDS: The number of context handles allowed per connection has been exceeded.
20440	zERR_PREV_DIR_AFTER_DATASTREAM: When parsing a data stream name, the previous dir symbols are not allowed.
20441	zERR_INVALID_PATH_FORMAT: The specified path format is not valid.
20442	zERR_CANT_WILDOPEN_A_DATASTREAM: An attempt was made to call zOpen on a data stream for the purpose of enumerating its contained file objects.
20443	zERR_NAMING_INCONSISTENCY: An internal naming inconsistency has occurred. The volume needs recovery.
20444	zERR_ZID_NOT_FOUND: The ZID is not found in the directory.
20445	zERR_LAST_STATE_UNKNOWN: The last consistent state of this file was that it did not exist.
20446	zERR_BAD_PATH_FORMAT: Path format specification is incorrect.

4.1.10 Name Typing Errors (20499)

20499	zERR_INVALID_NAME_TYPE: The specified name type is not valid.

4.1.11 Rename Errors (20500-20512)

20500	zERR_ALL_FILES_IN_USE: No files were renamed because all files were in use.
20501	zERR_SOME_FILES_IN_USE: Some files were not renamed because they were in use.
20502	zERR_ALL_FILES_READ_ONLY: No files were renamed because all files were read-only.
20503	zERR_SOME_FILES_READ_ONLY: Some files were not renamed because they were read only.
20504	zERR_ALL_NAMES_EXIST: No files were renamed because all destination names already existed.
20505	zERR_SOME_NAMES_EXIST: Some files were not renamed because the destination names already existed.
20506	zERR_NO_RENAME_PRIVILEGE: The caller does not have privileges to rename the file.
20507	zERR_RENAME_DIR_INVALID: The specified directory cannot be renamed because it either is the root directory of a volume, or because an attempt was made to rename it to a place in the tree below itself.

20508 zERR_RENAME_TO_OTHER_VOLUME: A file cannot be moved to another volume using rename.

20509 zERR_CANT_RENAME_DATA_STREAMS: Not allowed to rename a data stream.

20510 zERR_FILE_RENAME_IN_PROGRESS: The file is already being renamed by a different process.

20511 zERR_CANT_RENAME_TO_DELETED: Only deleted files can be renamed to a deleted state.

20512 zERR_RENAME_TO_OTHER_NAMESPACE: A file can not be renamed from one namespace to another.

4.1.12 Data Stream Errors (20550-20551)

20550 zERR_INVALID_DATA_STREAM: The specified data stream is invalid.

20551 zERR_CANT_MOD_DATA_STREAM_METADATA: It is illegal to modify the metadata for a data stream.

4.1.13 Semantic Agent Errors (20601-20602)

20601 zERR_INVALID_SA_HANDLE: The semantic agent handle ID is invalid.

20602 zERR_SA_HANDLE_TOO_SMALL: An attempt was made to allocate a semantic agent handle without providing a size that was big enough to hold the generic portion of the handle.

4.1.14 Direct FS I/O & DIO Errors (20650-20655)

20650 zERR_FILE_NOT_IN_DIO_MODE: The specified file is not in direct I/O mode.

20651 zERR_HOLE_IN_DIO_FILE: Direct I/O files cannot be sparse files with holes in them.

20652 zERR_BEYOND_EOF: Direct I/O files cannot be read beyond EOF.

20653 zERR_FILE_IN_DIO_MODE: File is in DIO mode.

20654 zERR_FILE_DETACHED: File is in DIO mode. Not used.

20655 zERR_DIO_BAD_PARAMETER: DIO bad parameter. Unit count is zero.

4.1.15 Namespace Errors (20700-20705)

20700 zERR_INVALID_NAMESPACE_ID: The specified namespace ID is invalid.

20701 zERR_UNABLE_TO_FIND_NAMESPACE: The specified namespace could not be found.

20702 zERR_INVALID_NAMESPACE_VERSION: The namespace version number is invalid.

20703 zERR_NAMESPACE_ID_IN_USE: The specified namespace ID is already in use.

20704 zERR_INVALID_PATH_SEPARATOR: The path separator requested is invalid in this namespace.

20705 zERR_VOLUME_SEPARATOR_NOT_SUPPORTED: This namespace does not support a volume separator.

4.1.16 Asynchronous Errors (20750-20751)

20750 zERR_BAD_ASYNCIO_HANDLE: The specified asynchronous I/O file handle is invalid.

20751 zERR_ASYNCIO_CANCELED: The operation failed because the asynchronous I/O request was canceled.

4.1.17 Encrypted Volume Errors (20798-20799)

20798 zERR_NICI_SUPPORT: Encrypted volume support libraries returned an error.

20799 zERR_ASYNCIO_CANCELED: The operation failed because the asynchronous I/O request was canceled.

4.1.18 Volume and Pool Errors (20800-20839)

20800 zERR_BAD_VOLUME_NAME: The specified volume name is invalid. A possible cause is that the name is too long.

20801 zERR_VOLUME_NOT_FOUND: The specified volume was unable to be located.

20802 zERR_DEACTIVATING_ADMINVOL: The NSS_ADMIN volume could not be deactivated.

20803 zERR_VOLUME_STATE_CHANGE_ABORTED: Had to abort the volume state change.

20804 zERR_DATA_MIGRATION_NOT_ENABLED: NSS does not currently support data migration.

20805 zERR_VOLUME_STATE_CHANGE_A_TO_M: Set by LSS if an attempt to go to the ACTIVE state was not completed because the volume should be placed into MAINTENANCE state.

20806 zERR_VOLUME_NOT_IN_MAINT_MODE: The given volume is not in MAINTENANCE mode.

20807 zERR_VOLUME_STATE_NOT_SUPPORTED: The volume does not support the state change requested.

20808 zERR_DUPLICATE_VOLUME_NAME: The volume name already exists.

20809 zERR_VOLUME_SCHEDULED_FOR_MAINT: The volume is already scheduled for MAINTENANCE.

20810 zERR_VOLUME_SHOULD_NOT_ACTIVATE: The volume should not be activated (LSS can return if it is corrupt or rebuilding).

20811 zERR_VOLUME_NOT_IN_ACTIVE_STATE: The specified volume is not in the ACTIVE state.

20812 zERR_POOL_NOT_FOUND: The specified pool name could not be found.

20813 zERR_POOL_STATE_INCOMPATIBLE: A volume change STATE has failed because the volume's pool is not in a compatible state. For example, if a pool is DEACTIVE and the volume wants to be ACTIVE, this error is returned. To fix the problem, the pool must be first placed in an acceptable STATE.

20814 zERR_RESERVED_VOLUME_NAME: The given volume name is a reserved name.

20815 zERR_BAD_VOLUME_NAME_CHARACTER: The volume name contains an invalid character.

20816 zERR_BAD_VOLUME_NAME_SIZE_LONG: The volume name is too long.

20817 zERR_BAD_VOLUME_NAME_SIZE_SHORT: The volume name is too short.

20818 zERR_BAD_VOLUME_NAME_UNDERSCORE: The volume name cannot start or end with an underscore.

20819 zERR_BAD_VOLUME_NAME_TWO_UNDERSCORES: The volume name cannot have two consecutive underscores.

20820 zERR_DUPLICATE_VOLUME_ID: The volume ID already exists.

20821 zERR_INVALID_VOLUME_ID: The volume ID is invalid.

20822 zERR_VOLUME_NOT_IN_DEACTIVE_STATE: The given volume is not in the DEACTIVE state.

20823 zERR_VOLUME_DELETION_MODE: The given volume is currently being deleted.

20824 zERR_VOLUME_CREATION_MODE: The given volume is currently being created.

20825 zERR_VOLUME_INVALID_MODE: The given volume is in an unknown mode.

20826 zERR_VOLUME_STATE_CHANGE_REQUESTED: The volume that an operation is running on is switching state. For example, when a LV is being deleted, the thread that does the deletion returns this error if the thread detects that the volume is changing state.

20827 zERR_VOLUME_STOP_REQUESTED: The volume that an operation is running on has detected that it must stop.

20828 zERR_VOLUME_ALREADY_UNLOADING: The volume is already being unloaded.

20829 zERR_VOLUME_RENAME_NOT_ALLOWED: The volume cannot be renamed. For example, the _ADMIN volume cannot be renamed.

20830 zERR_VOLUME_ACTIVE_ELSEWHERE: The volume is active on another server in the cluster. This is a 5.1 error. Use zERR_NWCS_VOLUME_IS_ACTIVE on later versions.

20831 zERR_VOLUME_READ_ONLY: The update operation failed on a read-only volume.

20832 zERR_VOLUME_BUSY_WITH_REQUEST: The operation cannot be completed because of a competing request.

20833 zERR_POOL_SHARED_NO_BROKER: The pool is marked SHARED but no cluster or broker software is loaded. It is unsafe to complete the requested operation because the pool might be in ACTIVE or MAINTENANCE state on another server.

20834 zERR_POOL_SHARED_STATE_UNKNOWN: Unable to detect if the pool is marked SHARED and no cluster or broker software is loaded. It is unsafe to complete the requested operation because the pool might be in ACTIVE or MAINTENANCE state on another server.

20835 zERR_DUPLICATE_POOL_NAME: The pool name already exists on the server.

20836 zERR_POOL_NOT_IN_ACTIVE_STATE: The given pool is not in ACTIVE state.

20837 zERR_POOL_RESERVATION_FAILED: When going activate, the pool failed to get a reservation within the MAL.

20838	zERR_VOLUME_IS_DEACTIVE: The operation could not be completed because the volume is deactive.
20839	zERR_POOL_IS_DEACTIVE: The operation could not be completed because the pool is deactive.

4.1.19 DSI and Adding Volumes Errors (20840-20849)

20840	zERR_IMPORT_DSI_SYMBOL_FAILED
20841	zERR_DSI_LOAD_FAILED
20842	zERR_DSIREG_RET2
20843	zERR_DS_NOT_SETUP
20844	zERR_DSIREG_FAILED
20845	zERR_DSI_LOGIN_FAILED
20846	zERR_ADD_TO_NDS_FAILED
20847	zERR_DEL_TO_NDS_FAILED
20848	zERR_REN_TO_NDS_FAILED
20849	zERR_VOL_UNAVAILABLE

4.1.20 Authorization Errors (20850-20871)

20850	zERR_NO_SET_PRIVILEGE: The caller does not have rights to modify metadata.
20851	zERR_NO_CREATE_PRIVILEGE: The caller does not have rights to create a file object.
20852	zERR_INVALID_AUTHORIZE_SPACE: Bad authorization space.
20853	zERR_INVALID_AUTHORIZE_MODEL: Bad authorization model.
20854	zERR_INVALID_AUTHORIZE_OPERATION: Bad operation passed to a function.
20855	zERR_AUTHORIZE_LOAD_FAILED: Failed to load part of the authorization system.
20856	zERR_TRUSTEE_NOT_FOUND: Unable to find the specified trustee.
20857	zERR_NO_TRUSTEES_FOUND: There were no trustees.
20858	zERR_NO_TRUSTEE_CHANGE_PRIVILEGE: The user does not have rights to change trustees on this file.
20859	zERR_ACCESS_DENIED: Authorization rules or file attributes prevent access to this file.
20860	zERR_NO_WRITE_PRIVILEGE: No write privileges have been granted for this file.
20861	zERR_NO_READ_PRIVILEGE: No read privileges have been granted for this file.
20862	zERR_NO_DELETE_PRIVILEGE: No delete privileges have been granted for this file.
20863	zERR_SOME_NO_DELETE_PRIVILEGE: Some of the files in a wildcard operation could not be deleted because no delete privileges were granted for those files.

20864	zERR_INVALID_AUTH_MODEL_VERSION: The authorization model being registered has an invalid version number.
20865	zERR_EXCEEDED_MAX_AUTH_SPACES: Exceeded the maximum number of authorization spaces.
20866	zERR_EXCEEDED_MAX_AUTH_MODELS: Exceeded the maximum number of authorization models.
20867	zERR_NO_SUCH_OBJECT: No such object was found in the naming services.
20868	zERR_CANT_DELETE_OPEN_FILE: Cannot delete the open file because it is flagged as being unable to be deleted while it is open.
20869	zERR_NO_CREATE_DELETE_PRIVILEGE: No create/delete privileges have been granted for this file.
20870	zERR_NO_SALVAGE_PRIVILEGE: No privileges to salvage this file.
20871	zERR_NO_SCAN_PRIVILEGE: No privileges to scan the directory or file.

4.1.21 NWCS Errors (20890-20898)

20890	zERR_NWCS_DUPLICATE_POOL_NAME: The pool name already exists within the cluster. Indicates that a local pool on one of the servers already is using the requested SHARED pool name.
20891	zERR_NWCS_DUPLICATE_VOLUME_NAME: The volume name already exists within the cluster. Indicates that a local volume on one of the servers already is using the requested SHARED volume name.
20892	zERR_NWCS_POOL_IS_ACTIVE: The pool is ACTIVE elsewhere in the cluster.
20893	zERR_NWCS_VOLUME_IS_ACTIVE: The volume is ACTIVE elsewhere in the cluster.
20894	zERR_NWCS_NOT_THE_OWNER: For attribute changing.
20895	zERR_NWCS_OPERATION_IN_PROGRESS: The operation is in progress on another cluster.
20896	zERR_NWCS_SHARE_VIOLATION: General (non-specific) NWCS error.
20897	zERR_NWC_NOT_A_MEMBER: The server is not a member of the cluster but is trying to access one of the shared resources.
20898	zERR_NWCS_DUPLICATE_IP_ADDRESS: The server is not a member of the cluster but is trying to access one of the shared resources.

4.1.22 Locking Errors (20900-20911)

20900	zERR_IOLOCK_ERROR: Tried to do read/write on a locked range of a file
20901	zERR_LOCK_ERROR: General lock error. Not used.
20902	zERR_LOCK_COLLISION: Tried to lock a range that was already locked.
20903	zERR_LOCK_WAITING: Timed out waiting for a lock.
20904	zERR_NONEXISTENT_LOCK: Tried to release a lock that doesn't exist.

20905 zERR_FILE_READ_LOCKED: Cannot grant read access to the file.

20906 zERR_FILE_WRITE_LOCKED: Cannot grant write access to the file.

20907 zERR_CANT_DENY_READ_LOCK: Cannot grant deny read access to the file.

20908 zERR_CANT_DENY_WRITE_LOCK: Cannot grant deny write access to the file.

20909 zERR_SELF_INFLICTED_COLLISION: Already have this lock.

20910 zERR_ALREADY_WAITING_FOR_LOCK: Already waiting for this lock.

20911 zERR_DEAD_LOCK: Cannot wait because a deadlock is detected.

4.1.23 Unicode Errors (20950-20955)

20950 zERR_UNICODE_INVALID_CONVERSION_TYPE: The Unicode conversion type is invalid.

20951 zERR_UNICODE_CONVERSION_ERROR: Error in converting to or from Unicode.

20952 zERR_UNICODE_INIT: Error initializing the Unicode subsystem.

20953 zERR_UNICODE_NON_MAPPABLE_CHAR: A nonmappable character was encountered while converting Unicode.

20954 zERR_EXCEEDED_MAX_CONVERSION_TYPES: Exceeded the maximum number of loadable conversion types.

20955 zERR_INVALID_UTF8_CHAR: Error converting UTF-8 to Unicode. Invalid UTF-8 sequence.

4.1.24 Link Errors (21000-21005)

21000 zERR_INVALID_LINK_TYPE: The specified link type is not supported.

21001 zERR_CANT_HARD_LINK_DATA_STREAMS: Not allowed to create hard links to or from a data stream.

21002 zERR_CANT_HARD_LINK_TO_DIRECTORY: Not allowed to create hard links to a directory or container.

21003 zERR_MUST_HARD_LINK_FROM_DIRECTORY: Not allowed to create hard links from a non-directory or container.

21004 zERR_CANT_HARD_LINK_TO_NON_FILE: Not allowed to create hard links to beasts not derived from a file.

21005 zERR_LINK_DEST_FILE_ALREADY_EXISTS: The new name for the link already exists.

4.1.25 NLM Registration Errors (21050-21058)

21050 zERR_MODULE_NAME_ALREADY_USED: The given NSS MODULE name is already in use.

21051 zERR_MODULE_NAME_NOT_FOUND: The given NSS module name could not be found.

21052 zERR_INVALID_MODULE_VERSION: The given MODULE has an invalid version number.

21053 zERR_INCOMPATIBLE_API_VERSION: The given MODULE has an incompatible API version number.

21054 zERR_INCOMPATIBLE_DEBUG_STATE: The given MODULE has an incompatible DEBUG version state.

21055 zERR_UNKNOWN_MODULE_TYPE: The given MODULE has an unknown module type.

21056 zERR_INVALID_REGISTRATION_TYPE: The given type is invalid for the item being registered.

21057 zERR_TYPE_ALREADY_REGISTERED: The given type is already registered.

21058 zERR_INCOMPATIBLE_MP_FLAG: The given MODULE has an incompatible MP state.

4.1.26 MASV Errors (21100)

21100 zERR_MASV_LABEL_ALREADY_SET: The label is already set.

4.1.27 Modify Volume Info Errors (21150-21151)

21150 zERR_SOME_ATTRS_NOT_CHANGED: On a modify of pool or volume enabled attributes, some were not changed.

21151 zERR_ALL_ATTRS_NOT_CHANGED: On a modify of pool or volume enabled attributes, all were not changed.

4.1.28 Feature Not Enabled Errors (21200-21215)

21200 zERR_EXTENDED_ATTR_NOT_ENABLED: Attempted to create extended attributes on a volume where the feature is not enabled.

21201 zERR_DATA_STREAMS_NOT_ENABLED: Attempted to create a named data stream on a volume where the feature is not enabled.

21202 zERR_DOS_METADATA_NOT_ENABLED: Attempted to write DOS metadata on a volume where the feature is not enabled.

21203 zERR_NETWARE_METADATA_NOT_ENABLED: Attempted to write NetWare® metadata on a volume where the feature is not enabled.

21204 zERR_MAC_METADATA_NOT_ENABLED: Attempted to write Macintosh metadata on a volume where the feature is not enabled.

21205 zERR_UNIX_METADATA_NOT_ENABLED: Attempted to write UNIX metadata on a volume where the feature is not enabled.

21206 zERR_HARD_LINKS_NOT_ENABLED: Attempted to create hard links on a volume where the feature is not enabled.

21207 zERR_TRANSACTIONS_NOT_ENABLED: Attempted to create user-level transactions on a volume where the feature is not enabled.

21208 zERR_USER_SPACE_RESTRICT_NOT_ENABLED: Attempedt to create user space restrictions on a volume where the feature is not enabled.

21209 zERR_COMPRESSION_NOT_ENABLED: Attempted to compress on a volume where the feature is not enabled.

21210 zERR_SPARSE_FILES_NOT_ENABLED: Attempedt to modify EOF on a file without modifying the physical size of the file.

21211 zERR_PHYSICAL_EOF_NOT_ENABLED: Attempted to extend the physical size of file independently of its logical size.

21212 zERR_DIRECT_IO_NOT_ENABLED: Attempted to use direct I/O on a volume where the feature is not enabled.

21213 zERR_MFL_NOT_ENABLED: Attempted to use MFL on a volume where it is not enabled.

21215 zERR_ENABLE_ENCRYPTED_VOLUME_NOT_ENABLED: Attempted to enable an encrypted volume when it is not enabled.

4.1.29 User Space Restriction Errors (21300-21303)

21300 zERR_ADDED_USER_TWICE: Tried to add the same user twice.

21301 zERR_NO_SUCH_USER: The requested user was not found in the tree.

21302 zERR_USER_SPACE_NOT_ENABLED: User space restrictions are not enabled.

21303 zERR_NOT_ENOUGH_USER_SPACE: Tried to allocate more than the restriction allows.

4.1.30 User Store Errors (21350-21351)

21350 zERR_FULL_NAME_NOT_FOUND: Name not found for a GUID.

21351 zERR_NEGATIVE_CACHE_ENTRY_FOUND: A negative entry was found in the cache.

4.1.31 Compression Manager Generated Errors (21400-21410)

21400 zERR_CM_ABORTED: Compression or decompression aborted.

21401 zERR_CM_INVALID_COMP_FILE_HEADER: Invalid compressed file header.

21402 zERR_CM_UNKNOWN_COMP_ALGO_VERSION: Unknown minor version was specified for the compression algorithm.

21403 zERR_CM_COMP_ALGO_ALREADY_REGISTERED: The compression algorithm is already registered.

21404 zERR_CM_CANT_DECOMPRESS: Cannot decompress file.

21405 zERR_CM_CANT_COMPRESS: Cannot compress file.

21406 zERR_CM_CORRUPT_COMPRESSED_FILE: The compressed file is corrupt.

21407 zERR_CM_COMP_ALGO_ERROR: Compression-algorithm-specific error.

21408 zERR_COMP_ALGO_NOT_REGISTERED: Compression algorithm is not registered.

21409 zERR_CM_INVALID_STREAM_HANDLE: Invalid stream handle.

21410 zERR_CM_INVALID_BUFFER_HANDLE: Invalid buffer handle.

4.1.32 Directory Quota Errors (21500-21505)

21500 zERR_ADDED_DIR_TWICE: Tried to add the same directory twice.

21501 zERR_NO_SUCH_DIR: The requested directory was not found in the tree.

21502 zERR_DIR_QUOTAS_NOT_ENABLED: Directory quotas are not enabled.

21503 zERR_NOT_ENOUGH_DIR_SPACE: Tried to allocate more than the quota allows.

21504 zERR_DIR_QUOTA_LATCH_ERROR: Could not get a latch on a cache entry.

21505 zERR_DIR_QUOTA_CACHE_ERROR: Error during cache add or lookup.

4.1.33 User Transaction Errors (21600-21606)

21600 zERR_TRANSACTION_DATA_TOO_LARGE: Trying to transaction too much data at once.

21601 zERR_TRANSACTION_LOG_FILE_NOT_WRITTEN: Could not write log file.

21602 zERR_NESTED_XACTIONS_NOT_IMPLEMENTED: Tried to use a nested transaction.

21603 zERR_TRANSACTION_LOG_FILE_OVERFLOW: The log file is full.

21604 zERR_VOLUME_NOT_TRANSACTED: Transactions are not supported on this volume.

21605 zERR_TRANSACTION_INVALID_STATE: Transaction not in correct state for operation.

21606 zERR_TRANSACTION_CROSSES_VOLUMES: Single transactions that operate on multiple volumes are not allowed.

4.1.34 Management File Errors (21700-21704)

21700 zERR_MODULE_NOT_FOUND: The module was not found.

21701 zERR_UNABLE_TO_GET_SET_PARAM_VALUE: An error was returned while getting the value of a set parameter.

21702 zERR_XML_IS_BAD: Bad XML found during parsing.

21703 zERR_XML_IS_INCOMPLETE: XML found without a terminating tag.

21704 zERR_UNABLE_TO_GET_ENOUGH_SIZE: The actual size created or expanded is less than what users ask for.

4.1.35 Data Migration Errors (21800-21803)

21800	zERR_DEMIGRATE_ALREADY_REGISTERED: Demigration function has already been registered.
21801	zERR_DEMIGRATE_NOT_AVAILABLE: No demigration function has been registered.
21802	zERR_DEMIGRATE_FAILED: Could not demigrate the file.
21803	zERR_MIGRATION_NOT_ALLOWED: Not allowed to migrate this file.

4.1.36 Semantic Agent Errors (22000)

22000	zERR_INVALID_OPENCREATE_MODE: Invalid mode passed in NCP 87 (create/open) or invalid open/create mode was passed to a traditional open/create function call.

4.1.37 CD-Specific Errors (22500-22512)

22500	zERR_CDDVD_CANT_READ_DISC
22501	zERR_CDDVD_INVALID_DISC
22502	zERR_CDDVD_INVALID_DIRECTORY_PTR
22503	zERR_CDDVD_INVALID_HFS_BTTREE_NODE
22504	zERR_CDDVD_INVALID_INFO_LENGTH
22505	zERR_CDDVD_ZID_NOT_FOUND_IN_HASH
22506	zERR_CDDVD_ASSOCIATED_FILE
22507	zERR_CDDVD_CANT_READ_ROOT_DIRECTORY
22508	zERR_CDDVD_INVALID_DESCRIPTOR_TAG
22509	zERR_CDDVD_CANT_FIND_VAT_ICB
22510	zERR_CDDVD_INVALID_PARTITION_MAP
22511	zERR_CDDVD_INVALID_LVID
22512	zERR_CDDVD_INVALID_ICB_TAG

4.1.38 DOSFAT/FAT32-Specific Errors (22600-22607)

22600	zERR_FAT_FILETYPE_NOT_SUPPORTED: Unsupported file type.
22601	zERR_FAT_CANT_INSTATIATE_FILE: Unable to export the file to NSS.
22602	zERR_FAT_ZID_NOT_FOUND_IN_HASH: Failed to find file in FAT hash.
22603	zERR_FAT_NOT_YET_IMPLEMENTED: FAT function not implemented.

22604	zERR_FAT_INVALID_DIR_ENTRY: Invalid FAT directory entry.
22605	zERR_FAT_ROOT_DIR_FULL: Root directory is full.
22606	zERR_FAT_INVALID_FAT_ENTRY: Invalid Fat entry.
22607	zERR_INVALID_CLUSTER_SIZE: Cluster size not power of 2.

4.1.39 NSS Java Interface Errors (22900-22999)

22901	zERR_JAVA_JNI_ERROR

4.1.40 SMS Tape LSS Errors (23000-23024)

23000	zERR_SMSTAPE_FIRST
23024	zERR_SMSTAPE_LAST

4.1.41 NFS Gateway LSS Errors (23100-23199)

23100	zERR_NFSGATEWAY_FIRST
23199	zERR_NFSTGATEWAY_LAST

4.1.42 Storage System B-Tree Errors (24800-24803)

24800	zERR_NO_NODE: No node at the requested block location.
24801	zERR_BAD_LOG_RECORD: Bad log record found.
24802	zERR_BEAST_TOO_BIG: The beast is too big to fit in the B-tree.
24803	zERR_TREE_LEAF_CORRUPT: Returned when the free size stored in a B-tree does not agree with the size of all the free chunks in the leaf.
24804	zERR_MISSING_BEAST: A required beast is missing. The administrator needs to verify the volume.

4.1.43 Storage System ZLOG Errors (24820-24831)

24820	zERR_ZLOG_BAD_CHECKSUM: ZLOG log record checksum error.
24821	zERR_ZLOG_FILE_TOO_SMALL: ZLOG file is too small.
24822	zERR_ZLOG_BAD_BLOCK_SIGNATURE: ZLOG file’s log block signature is invalid.
24823	zERR_ZLOG_BAD_RECORD_COUNT: ZLOG file’s log block log record count is invalid.

24824	zERR_ZLOG_BAD_RECORD_SIZE: ZLOG file's record size is invalid.
24825	zERR_ZLOG_BAD_LSN: ZLOG file's log record LSN is invalid.
24826	zERR_ZLOG_FILE_INIT_FAILED: ZLOG could not create ZLOG file during POOL initialize.
24827	zERR_ZLOG_BAD_BEAST_SIGNATURE: ZLOG beast's signature is invalid.
24828	zERR_ZLOG_UNSUPPORTED_BEAST_VERSION: ZLOG code does not support ZLOG Beast version in persistent storage.
24829	zERR_ZLOG_UNSUPPORTED_FILE_VERSION: ZLOG code does not support the ZLOG file version in persistent storage. If user does a clean shutdown with a previous `nss.nlm`, this error goes away. Otherwise, a reset is required.
24830	zERR_ZLOG_FILE_FULL: ZLOG file is full. It is too small for transaction rate.
24831	zERR_ZLOG_NO_MORE_RECORDS: No more ZLOG recovery information (not a USER error).

4.1.44 ZVL Errors (24838-24839)

24838	zERR_ZVL_UNSUPPORTED_BEAST_VERSION: The ZLSS volume locator code does not support the ZVL beast version in persistent storage.
24839	zERR_ZVL_BAD_BEAST_SIGNATURE: The ZLSS volume locator (ZVL) beast's signature is invalid.

4.1.45 ZFSVOL/ZLSSPOOL Volume Data Errors (24840-24849)

24840	zERR_ZFSVOL_BAD_CHECKSUM: ZFS volume data checksum error.
24841	zERR_ZFSVOL_AIPU_TOO_MANY_LVS: Too many logical volumes were encountered during the upgrade. Auto in-place upgrade returns this when upgrading to LVs if more LVs exist than expected.
24842	zERR_ZFSVOL_AIPU_LVDB_CORRUPTED: Logged volume data block number was incorrect during AIPU.
24843	zERR_ZFSVOL_AIPU_PHYSICAL_POOL: Physical pool went away during AIPU.
24844	zERR_ZFSVOL_NOT_A_ZLSS_VOLUME
24845	zERR_ZLSSPOOL_NOT_A_ZLSS_POOL
24846	zERR_ZFSVOL_AIPU_NOT_ACTIVE: Volume did not activate during LV AIPU.
24847	zERR_ZLSSPOOL_UPGRADE_POOL_FIRST: The ZLSS pool must be upgraded before the operation can be performed.
24848	zERR_ZLSSPOOL_NO_PHYSICAL_POOL: Could not find the physical pool.
24849	zERR_POOL_NOT_BIG_ENOUGH: Minimum ZLSS POOL size in 10 MBs.

4.1.46 Checkpoint Errors (24850-24853)

24850	zERR_CHECKPOINT_BAD_CHECKSUM: Checkpoint checksum error.
24851	zERR_CHECKPOINT_BAD_BLOCK_SIGNATURE: Checkpoint block signature is invalid.
24852	zERR_CHECKPOINT_BAD_BLOCK_SIZE: Checkpoint block size is invalid.
24853	zERR_CHECKPOINT_UNSUPPORTED_VERSION

4.1.47 Superblock Errors (24860-24868)

24860	zERR_SUPERBLOCK_BAD_CHECKSUM: Superblock checksum error.
24861	zERR_SUPERBLOCK_BAD_BLOCK_SIGNATURE: Superblock block signature is invalid.
24862	zERR_SUPERBLOCK_BAD_BLOCK_SIZE: Superblock block size is invalid.
24863	zERR_SUPERBLOCK_UNSUPPORTED_VERSION: Superblock code does not support the version in the superblock header.
24864	zERR_SUPERBLOCK_UNSUPPORTED_MEDIA: Media version not supported.
24865	zERR_SUPERBLOCK_MISMATCH: Two or more valid superblock headers do not match each other.
24866	zERR_SUPERBLOCK_UNDESIRED_LOCATION: A superblock is not in the desired mathematical location.
24867	zERR_SUPERBLOCK_NOT_ENOUGH: Not enough valid superblock headers.
24868	zERR_SUPERBLOCK_CORRUPTED: A valid superblock header has bad information in it.

4.1.48 Recovery Errors (24880)

24880	zERR_RECOVERY_TOO_MANY_UNCOMMITS: Recovery could not uncommit all transactions (that it needed to) before it reached the last checkpoint.

4.1.49 FSHooks Errors (24999)

24999	zERR_GENERIC_NSS_ERROR

4.2 Common Values

The common values of File System Services are described in the following sections:

- Section 4.2.1, "Effective Rights Values," on page 146
- Section 4.2.2, "Features Values," on page 146
- Section 4.2.3, "File Attributes Values," on page 148
- Section 4.2.4, "File Type Values," on page 149
- Section 4.2.5, "Get Info Mask Values," on page 150
- Section 4.2.6, "Inherited Rights Mask Values," on page 151
- Section 4.2.7, "Match Attributes Values," on page 152
- Section 4.2.8, "Modify Info Mask Values," on page 153
- Section 4.2.9, "Namespace ID Values," on page 154
- Section 4.2.10, "Name Type Values," on page 155
- Section 4.2.11, "Rename Flags Values," on page 155
- Section 4.2.12, "Requested Rights Values," on page 155
- Section 4.2.13, "Volume State Values," on page 157

4.2.1 Effective Rights Values

The bit values for effective rights are as follows:

0x0001 zAUTHORIZE_READ_CONTENTS
0x0002 zAUTHORIZE_WRITE_CONTENTS
0x0008 zAUTHORIZE_CREATE_ENTRY
0x0010 zAUTHORIZE_DELETE_ENTRY
0x0040 zAUTHORIZE_SEE_FILES
0x0080 zAUTHORIZE_MODIFY_METADATA
0x0100 zAUTHORIZE_SUPERVISOR

4.2.2 Features Values

Not all features of NSS are available or enabled for all volumes, pools, or LSSes. A particular LSS might support a feature but the administrator can choose not to enable that feature for a particular volume. For example, ZLSS supports salvage; however, for any particular ZLSS volume, the administrator can choose to disable salvage for that volume.

There are two fields that are defined to know which features are available: one that says which features are available and another to say which features are enabled. Both are 64-bit fields.

zFEATURE_SALVAGE

The ability to store deleted files, to enumerate them, salvage them, and to purge them.

zFEATURE_READONLY

If enabled, the object is read-only and cannot be written.

zFEATURE_COMPRESSION

The ability to store and manage files in a compressed format.

zFEATURE_USER_SPACE_RESTRICTIONS

The ability to keep track of the space used by each user.

zFEATURE_EXTENDED_ATTRIBUTES

Extended attributes can be stored and retrieved for a file.

zFEATURE_DATA_STREAMS

Can store and manage multiple data streams for a file including the Macintosh Resource Fork.

zFEATURE_DOS_METADATA

Can store and manage basic DOS metadata and attributes including Read-only, Hidden, System, and the basic DOS dates and time.

zFEATURE_NETWARE_METADATA

Can store and manage NetWare file metadata including all the NetWare attributes, creation time, archived time, modified time, accessed time, metadata modified time, primary namespace information, etc.

zFEATURE_MAC_METADATA

Can store and manage Macintosh metadata including resource forks and MAC finder information. Because resource fork must be stored, zFEATURE_DATA_STREAMS must also be set.

zFEATURE_UNIX_METADATA

Can store and manage UNIX metadata as required by the Novell® NFS product.

zFEATURE_HARD_LINKS

Allows multiple names in the same namespace for one file.

zFEATURE_TRANSACTION

User-level transactions can be used to ensure that either all updates occur or none of the updates occur.

zFEATURE_SPARSE_FILES

Sparse files can be stored and managed by the file system.

zFEATURE_PHYSICAL_EOF

A file can be extended or truncated without changing the logical end of the file.

zFEATURE_DIRECT_IO

The application can bypass the NSS cache and talk directly to the disk.

zFEATURE_PERSISTENT_FEATURES

The values defined in the these feature fields can be saved. It also means that some can be modified.

zFEATURE_VERIFY

The file system can check itself for inconsistencies.

zFEATURE_REBUILD

The file system has a built-in repair utility.

4.2.3 File Attributes Values

The file attributes parameter is a 64-bit mask that specifies the attributes associated with a file object.

zFA_READ_ONLY

This bit behaves like the DOS read-only file attributes. If set, the file might not be modified or deleted.

zFA_HIDDEN

This bit behaves like the DOS hidden file attribute. When using client utilities that honor this bit, the file is not visible.

zFA_SYSTEM

This bit behaves like the DOS system file attribute. It flags the file as being a system file.

zFA_EXECUTE

This bit is preserved by NSS, but is not used by the Z-APIs. The traditional NetWare Semantic Agent uses this bit.

zFA_SUBDIRECTORY

If this bit is set, the file is a directory container file object, the contents of which can be read using the zWildRead interfaces. Many of the file object types are directory container file objects, meaning that they contain other objects. File object types that are directory containers must have this bit set, and those that are not must not have this bit set.

zFA_ARCHIVE

This bit behaves like the DOS Archive attribute. If this bit is set, the file has been modified, but not yet archived. A backup utility should clear this bit after the file has been archived.

zFA_SHAREABLE

This bit is preserved by NSS, but is not currently used.

zFA_SMODE_BITS

This is a series of three bits. The file system reserves and stores these bits, but has no knowledge of their usage or interpretation. These bits are manipulated by clients, and are used as a place to store search modes associated with a file.

zFA_NO_SUBALLOC

This bit is preserved by NSS, but is not currently used, because suballocation is not supported.

zFA_IMMEDIATE_PURGE

If this bit is set, the file is not moved to the salvage system when it is deleted. A file that is deleted while this attribute is in effect can not be undeleted.

zFA_RENAME_INHIBIT

If this bit is set, the file cannot be renamed.

zFA_DELETE_INHIBIT

If this bit is set, the file cannot be deleted.

zFA_COPY_INHIBIT

If this bit is set, the file cannot be copied. It can be used by the client but not enforced by NSS.

zFA_REMOTE_DATA_INHIBIT

This bit is preserved by NSS, but is not currently used, because data migration is not yet supported.

zFA_COMPRESS_FILE_IMMEDIATELY

This bit triggers compression as soon as the file is closed.

zFA_DO_NOT_COMPRESS_FILE

This bit prevents the file from being compressed.

zFA_DATA_STREAM_IS_COMPRESSED_BIT

This bit indicates that the file is currently compressed.

zFA_CANT_COMPRESS_DATA_STREAM_BIT

This bit indicates that the data stream cannot be compressed.

zFA_ATTR_ARCHIVE

This bit behaves similar to the zFA_ARCHIVE bit, but it is only associated with the metadata of a file rather than the normal file data. If this bit is set, the file's metadata has been modified, but not yet archived. A backup utility should clear this bit after the file's metadata has been archived.

zFA_IS_LINK

This bit identifies this file object as representing either a symbolic link, a junction, or a URL link. The data stream for the file object contains the link information.

zFA_VOLATILE (0x80000000)

This bit prevents opportunistic locks from being returned. It also indicates that even if you have the file open for read-only, the contents can change. For example, if you read the same location from the file twice, you can get two different answers.

The end-of-file marker is not well defined for files marked with this attribute.

The following bits are supported in the traditional NetWare file system but are not yet supported in NSS. Support might be added for these bits as additional features are added to NSS.

OLD_PRIVATE_BIT
READ_AUDIT_BIT
WRITE_AUDIT_BIT
FILE_AUDITING_BIT
REMOTE_DATA_ACCESS_BIT
REMOTE_DATA_SAVE_KEY_BIT

The following bits are supported in the traditional NetWare file system but are deprecated for the zAPIs and are only available through the traditional NetWare APIs.

zFA_TRANSACTION

The zAPIs supply the transaction on the individual calls rather then setting a state in the open file. This lets one open file be used in multiple transactions.

4.2.4 File Type Values

The fileType parameter is used by zCreate to identify the type of file to be created. NSS supports several different types of file objects, the most basic of which is a file. Other types are used to represent system components such as storage pools, extended attributes. and volumes, or even the

NSS file system itself. The various components of the system can be enumerated by scanning the directories that contain these system component file objects and reading the information stored with those file objects.

zFILE_REGULAR

A normal data file or directory.

zFILE_EXTENDED_ATTRIBUTE

An extended attribute. Arbitrary metadata can be attached to a file.

zFILE_NAMED_DATA_STREAM

An additional streams attached to a file. Used to implement resource forks for Macintosh support.

zFILE_PIPE

(Future plan) Acts like a UNIX named pipe.

zFILE_VOLUME

A volume contains a set of files and directories.

zFILE_POOL

A pool contains storage that might be used by volumes.

4.2.5 Get Info Mask Values

The getInfoMask parameter is a 64-bit mask that specifies the file object information to be returned in the zInfo_s structure by zGetInfo. If no bits are set, no information is returned. Bits that don't make sense for the given object are ignored.

zGET_STD_INFO

Get the standard information about the file.

zGET_NAME

Get the name of the file object for the namespace and hard link used to locate the file object.

zGET_ALL_NAMES

Get the name of the file object for each namespace in which a valid name exists. Also, get the NameSpaceID of the name that is considered the original name for the file object. For a file object with multiple hard links, this only gets the names associated with the link used to find the file object.

zGET_PRIMARY_NAMESPACE

Get the nameSpaceID of the primary namespace for this file object's name.

zGET_TIMES_IN_SECS

Get the various time stamps associated with the file object in seconds. Although time is returned in seconds, the underlying system can only keep a resolution of 2 seconds.

zGET_TIMES_IN_MICROS

Get the various time stamps associated with the file object in micro-seconds. Though time is returned in micro-seconds, the underlying system may only keep a resolution of 1 or 2 seconds. Note that only zGET_TIMES_IN_SECS or zGET_TIMES_IN_MICROS but not both can be set.

zGET_IDS

Get the various eDirectory object IDs associated with this file object.

zGET_STORAGE_USED

Get information for this file object about the physical disk storage that is used.

zGET_BLOCK_SIZE

Get information about the file object's block size.

zGET_COUNTS

Get the file open count and hard link count information about the file object.

zGET_EXTENDED_ATTRIBUTE_INFO

Get summary information about extended attributes owned by this file object.

zGET_DATA_STREAM_INFO

Get summary information about data streams owned by this file object.

zGET_DELETED_INFO

Get deleted file information about the file. This includes the time in micro-seconds that the file was deleted and the user ID of the user who deleted the file.

zGET_MAC_METADATA

Get the file object's Macintosh metadata information.

zGET_UNIX_METADATA

Get the file object's UNIX metadata information.

zGET_EXTATTR_FLAGS

If the file object is an extended attribute, this bit returns the extAttrUserFlags.

zGET_VOLUME_INFO

If the file object is a volume, this bit returns the basic information about the volume.

zGET_VOL_SALVAGE_INFO

If the file object is a volume, this bit returns the salvage information for the volume.

zGET_DIR_QUOTA

If the file object is a directory, this bit returns the directory quota information for the directory.

NOTE: This bit requires zInfoB_s (page 113), which implies the zINFO_VERSION_B flag.

zGET_POOL_INFO

If the file object is a pool, this bit returns the basic information about the pool.

4.2.6 Inherited Rights Mask Values

The inheritedRightsMask parameter specifies the rights inherited while resolving a filename.

zAUTHORIZE_READ_CONTENTS

The user is allowed to read the contents of the files at this level in the tree and below.

zAUTHORIZE_WRITE_CONTENTS

The user is allowed to modify the contents of the files at this level in the tree and below.

zAUTHORIZE_CREATE_ENTRY

The user is allowed to create new files in this directory or below in the tree.

zAUTHORIZE_DELETE_ENTRY

The user is allowed to delete files in the directory or below in the tree.

zAUTHORIZE_ACCESS_CONTROL

The user is allowed to change the access control metadata in the tree or below.

zAUTHORIZE_SEE_FILE

The user can see the filenames in this directory and below.

zAUTHORIZE_MODIFY_METADATA

The user can modify the metadata for files at this level in the tree and below.

zAUTHORIZE_SUPERVISOR

The user is granted supervisor rights at this level in the tree and below.

zAUTHORIZE_SALVAGE

The user can salvage files at this level in the tree and below.

4.2.7 Match Attributes Values

The matchAttributes parameter is a bit mask that specifies what attributes a file must have to match in a wild card request. If no match attribute is given, all file objects match. The following masks can be combined by ORing to match on hidden, directory, and system attributes. For example, if you specify zMATCH_HIDDEN | zMATCH_SYSTEM, the matching file object must have both the attributes of HIDDEN and SYSTEM set (not HIDDEN or SYSTEM attributes).

zMATCH_HIDDEN

Matches if the hidden bit is set on the file.

zMATCH_NON_HIDDEN

Matches if the hidden bit is not set on the file. If both zMATCH_NON_HIDDEN and zMATCH_HIDDEN are set, nothing matches. If neither are set, both hidden and non-hidden files match.

zMATCH_DIRECTORY

Matches if the file object is a directory.

zMATCH_NON_DIRECTORY

Matches if the file object is not a directory. If both zMATCH_NON_DIRECTORY and zMATCH_DIRECTORY are set, nothing matches. If neither are set, both directories and other files match.

zMATCH_SYSTEM

Matches if a system file.

zMATCH_NON_SYSTEM

Matches if not a system file. If both zMATCH_NON_SYSTEM and zMATCH_SYSTEM are set, nothing will match. If neither are set, both system and non-system files match.

zMATCH_ALL

Matches all file objects in the given directory.

4.2.8 Modify Info Mask Values

The modifyInfoMask parameter is a 64-bit mask that specifies the file object information to be modified in the zInfo_s structure. If a bit is set in this bit mask, the corresponding field of modifyInfo is used to modify the file. All other fields in the zInfo_s structure (for which the bits are not set) are ignored. If a bit doesn't make sense for a particular file object, it is ignored.

zMOD_FILE_ATTRIBUTES

Modify the file object's attributes.

zMOD_CREATED_TIME

Modify the file object's creation time.

zMOD_ARCHIVED_TIME

Modify the file object's last archived time.

zMOD_MODIFIED_TIME

Modify the file object's last modified time.

zMOD_ACCESSED_TIME

Modify the file object's last accessed time.

zMOD_METADATA_MODIFIED_TIME

Modify the file object's last metadata modified time.

zMOD_OWNER_ID

Modify the file object's owner ID.

zMOD_ARCHIVER_ID

Modify the file object's archiver ID.

zMOD_MODIFIER_ID

Modify the file object's modifier ID.

zMOD_METADATA_MODIFIER_ID

Modify the file object's metadata modifier ID.

zMOD_PRIMARY_NAMESPACE

Modify the file object's primary namespace ID.

zMOD_DELETED_INFO

Modify the file object's deleted info. If the file object is not deleted, this is ignored.

zMOD_DELETED_ID

Modify the file object's deleted ID info only. If the file object is not deleted, this is ignored.

zMOD_MAC_METADATA

Modify the file object's Macintosh metadata.

zMOD_UNIX_METADATA

Modify the file object's UNIX metadata.

zMOD_EXTATTR_FLAGS

If the file object is an extended attribute, modify the value of extAttrUserFlags.

zMOD_VOL_FEATURES

If the file object is a volume, enable or disable the features associated with this volume.

zMOD_VOL_NDS_OBJECT_ID

If the file object is a volume, modify the value of the eDirectory object ID that identifies this volume in the tree.

zMOD_VOL_MIN_KEEP_SECONDS

If the file object is a volume, modify the value of salvage.minKeepSeconds. This is the minimum number of seconds that a deleted file remains on the volume before it becomes a candidate for auto purging.

zMOD_VOL_MAX_KEEP_SECONDS

If the file object is a volume, modify the value of salvage.maxKeepSeconds. This is the maximum number of seconds that a deleted file remains on the volume before it must be auto purged.

zMOD_VOL_LOW_WATER_MARK

If the file object is a volume, modify the value of salvage.lowWaterMark. This is an integer percentage value. If the percentage of free space on the volume falls below this value, auto purging is initiated.

zMOD_VOL_HIGH_WATER_MARK

If the file object is a volume, modify the value of salvage.highWaterMark. This is an integer percentage value. If the percentage of free space on the volume rises above this value, auto purging is terminated.

zMOD_VOL_QUOTA

Allows you to change the quota on a volume.

4.2.9 Namespace ID Values

The following namespaces are defined:

zNSPACE_DOS

This namespace supports 8.3 names. All names are stored in uppercase and use standard DOS file naming rules.

zNSPACE_MAC

This namespace supports 31-byte Macintosh filenames. The names are case preserving but not case sensitive. The names follow the standard Macintosh file naming rules.

zNSPACE_UNIX

This namespace supports 256-byte names. All names are stored in a case-preserving manner, and names are case sensitive. The names follow the standard UNIX file naming rules.

zNSPACE_LONG

This namespace supports 256-byte names. All names are stored in a case-preserving manner, but names are not case sensitive. The names use the standard file naming rules of Windows* 95, Windows NT*, and OS/2*.

zNSPACE_DATA_STREAM

This namespace is used for data streams. It is supported for names whose name type is zNTYPE_DATA_STREAM. It has the same name semantics as the long namespace.

zNSPACE_EXTENDED_ATTRIBUTE

This namespace is used for extended attributes. It is supported for names whose name type is zNTYPE_EXTENDED_ATTRIBUTE. It has the same name semantics as the those in the current NetWare extended attributes.

4.2.10 Name Type Values

The following name types are defined:

zNTYPE_FILE

This name type is used for all standard files and subdirectories.

zNTYPE_DATA_STREAM

This name type is used for named data streams that use the data stream namespace to store their name.

zNTYPE_EXTENDED_ATTRIBUTE

This name type is used for named data streams that use the extended attribute namespace to store their name.

zNTYPE_DELETED_FILE

This name type is used for files and subdirectories that have been deleted, but that are still stored in the salvage system, from which they can be undeleted.

4.2.11 Rename Flags Values

This is a bit mask that identifies various modes to be communicated when renaming a file object.

zRENAME_ALLOW_RENAMES_TO_MYSELF

If this bit is set, file objects can be renamed to the same name without generating an error. If this bit is not set, an attempt to rename a file object to itself generates an error.

zRENAME_THIS_NAME_SPACE_ONLY

If this bit is set, then the file object is only renamed in the specified namespace, and it cannot be moved to another destination directory. The name remains unchanged in the other namespaces. If this bit is not set, the file object is renamed in all applicable namespaces, and it can be moved to a different directory.

4.2.12 Requested Rights Values

This is a bit mask identifying requested access rights and behaviors to be associated with a key. These bits are not stored with the file. Instead, they are temporary bits associated with an open instance of the file.

zRR_READ_ACCESS

Read access is requested on the open file handle.

zRR_WRITE_ACCESS

Write access is requested on the open file handle.

zRR_DENY_READ

Deny read access is requested on the open file handle. No other readers should be allowed to have the file open.

zRR_DENY_WRITE

Deny write access is requested on the open file handle. No other writers should be allowed to have the file open.

zRR_SCAN_ACCESS

Scan access to file object. You can use zWildRead to scan the files, subdirectories, and attributes of a file or directory.

zRR_DELETE_FILE_ON_CLOSE

When the file is closed, it is deleted.

zRR_FLUSH_ON_CLOSE

Flush the user data and metadata for the file when it is closed. If the file is closed by the system because of a disconnect, it is still flushed.

zRR_PURGE_IMMEDIATE_ON_CLOSE

Works with zRR_DELETE_FILE_ON_CLOSE to keep the file from being placed in the salvage system when it is deleted on close.

zRR_DIO_MODE

Opens a file in direct I/O mode. Replaces having a separate call to change to direct I/O mode.

zRR_DONT_READ_AHEAD_ON_OPEN

Opens a file only. Does not read ahead data blocks (default value is 2) for any open file, if the file size is > 0. Used by SMS for performing backup, when parallel files are open.

zRR_CREATE_WITHOUT_READ_ACCESS

When this bit is set, a file object can be created if read access was requested, even if the read access is not allowed.

zRR_CANT_DELETE_WHILE_OPEN

While the file is open with this bit set, it cannot be deleted.

zRR_DONT_UPDATE_ACCESS_TIME not implemented

When the file is opened, the metadata last accessed time should not be updated. Intended to let the clients access the icon information without causing a change in the file metadata that eventually needs to be written to disk.

zRR_ENABLE_IO_ON_COMPRESSED_DATA_BIT

Rather than reading the decompressed version of the data stream, read the compressed version of the data stream. zERR_DENY_WRITE must be set for this compression flag to work.

zRR_LEAVE_FILE_COMPRESSED_BIT

Keep the compressed version of the file even after decompressing it.

zRR_READ_ACCESS_TO_SNAPSHOT

Return information from the snapshot and not the original file object.

The following bits are supported in the traditional NetWare file system but are not yet supported in NSS: FILE_READ_THROUGH_BIT, FILE_WRITE_THROUGH_BIT, ALWAYS_READ_AHEAD_BIT, NEVER_READ_AHEAD_BIT, NO_RIGHTS_CHECK_ON_OPEN_BIT ALLOW_SECURE_DIRECTORY_ACCESS_BIT

4.2.13 Volume State Values

This is a value identifying the state associated with this volume.

zVOLSTATE_UNKNOWN

The state of this volume is unknown.

zVOLSTATE_DEACTIVE

This volume has been deactivated and is not available for use.

zVOLSTATE_MAINTENANCE

This volume is in a maintenance state. It is either being checked or repaired and is temporarily not available for use.

zVOLSTATE_ACTIVE

This volume is active and available for use.

4.3 Data Types

The following list describes all of the basic data types associated with File System Services:

ADDR

An ADDR is a simple typedef defining an unsigned integer value that is capable of holding a pointer. Never passed over the wire.

BYTE

A BYTE is a simple typedef defining an unsigned 8-bit value.

GUID_t

A global unique identifier, GUID, is a simple typedef defining a 128-bit identifier that is globally unique.

Key_t

A Key_t is a simple typedef defining a signed 64-bit value. Used for storing keys.

LONG

A LONG is a simple typedef defining an unsigned 32-bit value.

NINT

A NINT (unsigned Native integer) is a simple typedef defining an unsigned integer value that is a minimum of 32 bits in length. If it is larger than 32 bits, no more than 32 bits of a NINT is used. On a 32-bit processor, a NINT is 32 bits and on a 64 bit processor, a NINT is 64 bits. This is to improve interactions with the memory cache. On the wire, NINTs are passed as LONGs.

QUAD

A QUAD is a simple typedef defining an unsigned 64-bit value.

SBYTE

An SBYTE is a simple typedef defining a signed 8-bit value.

SWORD

An SWORD is a simple typedef defining a signed 16-bit value.

SLONG

An SLONG is a simple typedef defining a signed 32-bit value.

SNINT

An SNINT (signed Native integer) is a simple typedef defining a signed integer value that is a minimum of 32 bits in length. If it is larger than 32 bits, no more than 32 bits of an SNINT is used. On a 32-bit processor, a SNINT is 32 bits and on a 64-bit processor, an SNINT is 64 bits. This is to improve interactions with the memory cache. On the wire, SNINTs are passed as SLONGs.

SQUAD

An SQUAD is a simple typedef defining a signed 64-bit value.

STATUS

The STATUS type is a simple typedef defining a signed integer value large enough to handle the full range of values returned by any of the APIs. For a description of the various values assigned to this type, refer to the description of STATUS Codes.

Time_t

This is the standard UTC time as defined by the C language.

unicode_t

Unicode is fast becoming an international standard for representing character sets. It uses 16 bits for each character.

UserID_t

The GUID that uniquely identifies a user on the system.

Utf8_t

A regular string except the characters are encoded using the UTF-8 rules.

VolumeID_t

The GUID that uniquely identifies a volume.

WORD

A WORD is a simple typedef defining an unsigned 16-bit value.

Xid_t

The Xid_t is a simple typedef used for storing transaction IDs. It is defined as being an unsigned value large enough to store a transaction ID.

Zid_t

The Zid_t is a simple typedef used for storing ZID values. It is defined as a 64-bit unsigned value.

zInfo_s

A structure that is used to return or modify detailed information about a file object (see zInfo_s (page 108)

Glossary

This section defines terms that are commonly used in connection with File System Services.

authorization system. Provides access authorization for file objects in NSS. The NSS system is designed so that the authorization system is modular. NSS is shipped with a default traditional NetWare® trustee-based authorization system. Other authorization systems might be provided in the future. Every volume has exactly one authorization system associated with it. The loaded authorization systems available on a server and information about which volumes use which authorization systems can be enumerated by reading the directory and file information contained in the NSS_ADMIN volume directory hierarchy of each server. The NULL authentication system always gives permission.

connection. When a client is connected to and authenticated on a server, a connection is associated with that client. Callers of the NLM interfaces are also considered to be clients, even though they are local on the server. Each server has a built-in internal system connection which is preauthenticated with all supervisor rights to the server. An NLM has the option of establishing its own authenticated connection on behalf of remote clients (by using eDirectory™ login interfaces), or it can use the internal default system connection.

context. The concept of a current directory is used in most client environments. Rather than starting at the root of a volume's tree, a key (obtained by opening a directory in the volume) can be used to identify a current directory context.

data streams. A single file can have multiple streams of data associated with it. The various data streams can be opened and read independently of each other. The primary data stream of a file is directly associated with the file object itself. All data files, by default, have a primary data stream. All additional secondary data streams are identified by a Unicode data stream name which further qualifies the file name. A NULL data stream name identifies the primary data stream. An example of a data stream would be the Macintosh Resource Fork, MAC_RF, which is a separate stream of data (containing icons) that is associated with a Macintosh file. There are two name types supported for data streams. The data stream names follow the naming semantics used by Windows 95/98/NT.

direct I/O. Normally, all reads and writes to a file pass through a file system cache buffer. Some applications, such as database programs, might need to do their own caching. With direct I/O, an application can bypass the cache buffer and do reads and writes directly to the file itself.

effective rights. Effective rights are the maximum actual rights that a specific client (an authenticated eDirectory object) is allowed to have to a specific file object, not including the file-specific rights as defined by the file attributes. The authorization system associated with a volume controls how effective rights are granted, restricted and inherited, etc., within that volume. When a request is made to perform an operation on a file, the authorization system compares the rights needed to perform the operation against the effective rights to determine if the operation should be allowed.

extended attributes. Extended attributes are very similar to named data streams, except that they use the NetWare extended attribute naming semantics. Extended attributes also have their own object type, because they contain a unique piece of metadata that is not contained in the data stream object. This metadata is a user-defined 32-bit value that is stored with each extended attribute and that is opaque in meaning to NSS.

file attributes. All file objects have basic file attributes associated with them. Among other things, these file attributes indicate whether the file is read-only, or whether it may be deleted, purged or copied. These file attributes are completely independent from the authorization system and take

precedence over the effective rights granted via that authorization system. For example, effective rights might allow write access to a file; but if the file attributes for the file indicate the file is a read-only file, writes are not allowed on the file. Likewise, if the file attributes indicate that the file has the delete inhibit bit set, the file cannot be deleted, regardless of the effective rights a user has.

file object. The file system is made up of many different types of files. A file object is an actual instance of a file type.

file type. The file system is made up of many different types of file objects, the most basic of which is a file. Other types are used to represent system components such as volumes, extended attributes, and storage pools, or the NSS file system itself. The various components of the system can be enumerated by scanning the directories that contain these system component file objects and reading the information stored with those file objects.

hard link. Hard links let NSS users create multiple names for the same file object. These names can be in different directories as long as all the directories reside on the same volume. Hard links to directories, data streams, and extended attributes are not allowed. NSS keeps a count of how many links a file has and does not delete a file until all names for the file have been deleted (the link count goes to zero).

junction. A junction is a special type of file that, instead of having data, contains a pathname to the root of another volume. The name parser (either on the client or in the File System API) finds the new volume and continues parsing the name on the new volume, which lets several volumes appear as one volume to the user.

keys. To do anything with the file system, you must have a key. Keys are used to read and write files, open files, get information about files, scan directories, rename files, identify current directories, etc.

A key is a 64-bit random number that hashes to a door, which points to a file handle structure that can represent an open file or a directory with a search map for enumerating the directory, a context for beginning a pathname lookup, or all of the above. When a file is forcibly closed by the system, the door can still exist but is now broken and cannot be used for further operations, which makes local and remote operations look the same.

namespace. Namespaces control the syntax of naming, such as which characters are legal in file names and in path separators, how big the name can be, and whether case is significant and/or preserved. The NSS file system is designed so that all namespaces are loadable and replaceable. NSS is shipped with namespace modules for DOS and LONG (OS2, NT/WIN95), Macintosh, UNIX, data stream and extended attribute support.

Every volume must have one or more namespaces associated with it. If a volume has more than one namespace associated with it, valid file names are maintained on that volume for all of its namespaces. If a file name is legal in more than one namespace, it is stored only once and shared by the namespaces. The loaded namespaces available on a server and information about which volumes use which namespaces can be enumerated by reading the file object information contained in the NSS_ADMIN volume directory hierarchy of each server.

The data stream and extended attribute namespaces are not used for standard filenames on volumes. They are used only for naming data streams or extended attributes.

name type. Every directory file can store multiple types of named objects, which are identified by a name type. A name type is simply used to logically group similar types of named objects together in a directory. For example, a standard directory container can contain both named files and named data streams. Both types of names are stored in the same directory, but they are logically grouped together so that each type of name is distinguishable from the other. How name types are dealt with in the physical storage is left up to each individual storage system.

All normal file and directory names are stored with the zNTYPE_FILE name type. All data stream names are stored with either the zNTYPE_DATA_STREAM or with the zNTYPE_EXTENDED_ATTRIBUTE name type. Deleted files in the salvage system are stored with the zNTYPE_DELETED_FILE name type and other features can use additional name types.

open file structure. When a file is opened or created, an internal data structure called an Open File Structure is allocated that represents that state of that open file. The user gets back a key to the open file structure that must be used for all subsequent accesses to the file (reads, writes, wild-reads, and getting and modifying file metadata).

partial path resolution. Alhough not exposed by the File System functions, partial path resolution is where the server returns to the client when it encounters a junction or symbolic link while parsing the path. The client can then continue parsing the path on the appropriate server, which lets the client directly talk to the server that manages the file object named by the path without going through an intermediary server.

pathname, fully qualified. A pathname is made up of the name of a file object, preceded by optional directory container names and directory separators. All pathnames are relative to some starting base location, such as a directory container file object or the root of a volume. A fully qualified pathname is a pathname that includes an eDirectory tree name or it is a pathname that starts with a volume name.

requested rights. Requested rights apply only to the files about to be opened. They are not the effective rights a client has on a file, but they are the rights that a client is requesting to have for a particular open instance of the file. For example, a client may choose to open the same file twice—the first time with read access and the second time with write access. The client would be given two keys but would be able to only read with the first key and able to only write with the second one. The rights granted to a key are stored with the open file structure.

When a request is made to open a file, the authorization system compares the client's requested rights parameter with the client's effective rights to the file and the additional file permission attributes (for example: read, write, deny-read, and deny-write). If the effective rights do not allow the requested rights to be granted, the open is denied.

salvage system. Unlike most file systems, NetWare provides a well-defined way of recovering "deleted" files. Rather than immediately freeing the disk blocks for a deleted file, NSS marks the name as deleted. Only when the space of the deleted file is needed is the file actually purged from the system. By using zMODE_DELETED to augment the source namespace on zRename, you can recover deleted files. By using zNTYPE_DELETED_FILE for the name type in zOpen, the deleted files can be viewed.

storage pool. Unlike previous versions of NetWare, the NSS volumes do not represent physical storage. An NSS storage pool is simply a collection of disk segments that provide physical storage to one or more NSS volumes. Volumes are logical entities that participate in naming and locating files. Storage pools provide storage for the volumes that reside on them.

storage system. The storage system provides basic raw file storage to NSS. It is a flat storage system, where all files are identified by a single identifier called a ZID. The storage system is also used to control how directory contents are organized and stored/retrieved. The NSS file system is designed so that the storage system is replaceable and so that several of these loadable storage systems (LSSes) can be active on the server at once. NSS is shipped with a default journal-based storage system, also known as the Z Loadable Storage System (ZLSS). Other storage systems (such as CD-ROM or DOS/FAT) are also provided. Every storage pool has exactly one loadable storage system associated with it. The loaded storage systems available on a server and information about which storage pools use which storage systems can be enumerated by reading the file object information contained in the NSS_ADMIN volume directory tree for each server.

symbolic links. A symbolic link is a special type of file whose data is a pointer to another file. The pointer is specified in the form of a path string identifying the new file. This new file does not need to be on the same volume or even on the same server. When a symbolic link is encountered while parsing a path, the link is followed and the path parsing continues with the newly specified path.

task. The purpose of tasks is to maintain separate environments for each of the applications, while sharing a single authenticated connection. When a client workstation connects to a server, it typically establishes a single authenticated connection. However, the client might be running multiple applications, each of which needs to maintain separate open files, current directory context information, etc. Each client application has a taskID associated with it; and when an application terminates, all keys associated with the task can be reclaimed without affecting those in use by other tasks on the same connection. An NLM has the option of using tasks in any way it finds necessary to meet its own needs. If the NLM is providing services to other clients, it can use real taskID values. The file system API defines two special values for the task ID.

zSAME_TASK specifies to use the same task as the key.

zROOT_TASK corresponds to the task 0 in traditional NetWare and is created when the connection initializes with NSS.

transactions. A transaction allows a set of operations be treated as a single atomic operation where either all are completed or none are completed. Most functions can be, but are not required to be, bound to a transaction. The xid (transaction ID) parameter identifies whether or not the function is associated with a transaction. If xid contains the value zNILXID, the function is not a part of a transaction; otherwise, xid contains the ID of an in-progress transaction to be associated with the function. Functions like zCreate take a transaction ID because they can change metadata about a file, including creating or destroying a file, which permits backing out changes to the file system as part of a user transaction. Even though an interface might include a xid parameter, it does not mean that transactions are currently supported for that function. Instead, it is provided for future features so interfaces do not need to change or be added.

Transactions can be nested by making them part of another transaction. The original transaction is called the parent and the new transaction is called the child. The child can then create its own child transactions. A transaction cannot commit until all of its children are committed. Locks held by a child are passed back to the parent when the child commits.

Locks held by a transaction cannot be released until the transaction commits. If it is a child transaction, the locks are not released until all of its parents have committed. When the transaction is committed, all the locks are released.

userID. All operating systems have a mechanism for identifying users on the system. In NetWare, the user's NDS Object ID is used (a 128-bit value); in other operating systems, different IDs are used. The userID uniquely identifies a user to the system.

UTF8. UTF-8 is a variable-length byte stream encoding of Unicode that is used by Java* and XML. Any Unicode character can be represented in UTF-8. It has numerous advantages over regular Unicode for serial communication, the best of which is that C ASCII strings have the same representation.

volume. An NSS volume is different from the traditional NetWare volume of the past. NSS volumes do not represent physical storage; they represent only a logical collection of files. A logical NSS volume is stored in a physical NSS storage pool. Volumes are made up of a rooted collection of file objects arranged in a directory hierarchy. They can be migrated from storage pool to storage pool and server to server as access needs and space availability change. They can also be replicated in multiple locations for more efficient distributed access.

Volumes participate in the naming and location of files as follows. If a pathname is an NDS fully qualified pathname (starting with \\\), the name parsing begins at the root of the NDS tree and works its way down the NDS containers until it encounters the name of an NDS volume object or an NDS junction object. If a pathname begins with a single slash (\), the path does not specify a volume. However, it does indicate that the remainder of the path is relative to the root of the volume identified using a different mechanism. For the LONG and DOS namespaces, if a pathname begins with a name followed by a colon (:), the name preceding the colon must identify a volume object that is local on the server that is parsing the name. Through other NSS algorithms, an appropriate instance of the volume is located and the rest of the unparsed pathname is used to identify the file object within the volume.

Every server has a default NSS_ADMIN volume that stores system configuration information in its directory hierarchy. System objects such as volumes, storage pools, namespaces, storage systems and authorization systems can be enumerated by reading the directory container file objects in this volume.

wildcards. Only the DOS and LONG namespaces support wildcards. When using these namespaces, some functions allow wildcard characters to allow multiple files to be acted upon by calling a single function. The legal wildcard characters for DOS and LONG namespaces are asterisk (*) and a question mark (?).

XID. NSS employs a flat storage system, where all files are identified by a single identifier called a ZID. A ZID uniquely identifies a file within a given volume. In order to uniquely identify the path used to get to a file, a parent ZID and name identifier are also required because hard links make it possible for a file to have more than one parent, which creates a need to identify which parent was used to get to the file.

zRootKey. Returns an initial key for a particular connection. To use the system connection, an NLM passes the zSYS_CONNECTION value to zRootKey for the connectionID parameter; otherwise the NLM must pass in a valid connectionID which was returned by the login interfaces.

A Revision History

The following table outlines all the changes that have been made to the 64-Bit File System Services (FS64) documentation (in reverse chronological order).

December 2011	Added zZIDDelete2 and zZIDRename2 standard functions in the Chapter 2, "Functions," on page 17.
May 2010	Added zNewConnectionWithBlob section in the Chapter 2, "Functions," on page 17.
	Changed the file attribute to create a link from zFA_LINK to zFA_IS_LINK.
December 2008	Added zNewConnection section in the Chapter 2, "Functions," on page 17.
October 17, 2007	Added the Section 1.7, "OES 2 Functions," on page 15 section and marked every function that is supported on Novell® Open Enterprise Server 2 (OES 2).
	Added zGetInfoByName (page 46), zInfoGetFileName (page 50), and zModifyInfoByName (page 57).
	Added the zRR_READ_ACCESS_TO_SNAPSHOT bit to Section 4.2.12, "Requested Rights Values," on page 155.
	Added note to the preface of Chapter 1, "Concepts," on page 9 stating that only NSS volume names are supported. File System Services does not recognize eDirectory names.
	Revised the information in Section 1.2, "Pathname Resolution," on page 10 to reflect that File System Services does not recognize eDirectory names. Also, removed the former 1.2.3 section on Relative Fully Qualified Paths.
October 11, 2006	Fixed spelling of QUAD in the time (page 104) structure.
March 1, 2006	Fixed spelling of zCREATE_TRUNCATE_IF_THERE.
	Made minor editing changes.
October 5, 2005	Transitioned to revised Novell documentation standards.
March 2, 2005	Added zPoolInfo_s (page 119) and zVolumeInfo_s (page 122).
	Added a new description of FS64 to the first part of "Concepts" on page 9.
	Updated the information on "Match Attributes Values" on page 152.
June 9, 2004	Added three return values for encrypted volumes (see 20798 and 20799 in Section 4.1.17, "Encrypted Volume Errors (20798-20799)," on page 134 and 21215 in "Feature Not Enabled Errors (21200-21215)" on page 139).
	Added colon to book name.

February 18, 2004	Added declarations for the zMacInfo_s (page 118) and zUnixInfo_s (page 121) structures. Updated the description of the match parameter in zWildRead (page 74). Deleted references to ROOT_TASK.
October 8, 2003	Updated the syntax of zLink (page 51) and added the last parameter, renameFlags.
June 2003	Updated the syntax of zZIDRename (page 85).
March 2003	Added documentation for zGetEffectiveRights (page 41), zZIDDelete (page 79), zZIDOpen (page 83), zZIDRename (page 85), and "Effective Rights Values" on page 146. Added a new bit to "Get Info Mask Values" on page 150. Added the zInfoB_s (page 113) structure. Updated the description of the connectionID parameter in zRootKey (page 69). Updated the description of the nameSpace parameter in zCreate (page 27) and zOpen (page 61).
September 2002	Updated all Return Values in Chapter 4, "Values," on page 125. Updated the section on "keys" on page 162 in the Glossary. Also, updated the Remarks section on keys in zRootKey (page 69). Changed the description of several values in "Match Attributes Values" on page 152. Added a note that zERR_DENY_WRITE must be set for zRR_ENABLE_IO_ON_COMPRESSED_DATA to work (see "Requested Rights Values" on page 155).
May 2002	Updated documentation for zInfo_s (page 108).
February 2002	Added documentation for Real Time Data Migration functions (Section 2.2, "Data Migration Functions," on page 89). Added documentation for zEnumerate (page 38). Updated the definition of retEndingOffset in zGetFileMap. Updated the description of "Match Attributes Values" on page 152. Removed zSAME_TASK and zROOT_TASK as possible taskID values. These values were never implemented. Changed all occurrences of zINFO_VARIABLE_DATA_SIZE to zGET_INFO_VARIABLE_DATA_SIZE in zInfo_s (page 108). Changed zCHAR_UTF8 to zPFMT_UTF8 in zWildRead (page 74).
September 2001	Added descriptions to graphics.
June 2001	Added zMOD_VOL_QUOTA to "Modify Info Mask Values" on page 153. Updated tables.

February 2000	Added zFA_VOLATILE attribute to the "File Attributes Values" on page 148 section. Change fileAttributes of zCreate (page 27) to a QUAD. Also changed the description of zCREATE_DELETE_IF_THERE in Parameters section. Changed the description of characterCode in zWildRead (page 74). Changed a padding field to dataShreddingCount in zInfo_s (page 108) and added description in Fields section.
September 2000	Moved zInfo_s (page 108) information from previous Section 4.3, "Data Types," on page 157 section to Chapter 3, "Structures," on page 93. Combined into one section three previous values sections: Section 4.1, "Return Values," on page 126, Section 4.2, "Common Values," on page 146, and Section 4.3, "Data Types," on page 157.
July 2000	Added new section on Section 1.5, "Header Files," on page 13 and changed the definition of timestamps in the preface.