



System i
Files and file systems
Integrated file system

Version 5 Release 4





System i
Files and file systems
Integrated file system

Version 5 Release 4

Note

Before using this information and the product it supports, read the information in “Notices,” on page 133.

Seventh Edition (February 2006)

This edition applies to version 5, release 4, modification 0 of IBM i5/OS (product number 5722-SS1) and to all subsequent releases and modifications until otherwise indicated in new editions. This version does not run on all reduced instruction set computer (RISC) models nor does it run on CICS models.

© Copyright International Business Machines Corporation 1999, 2006.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Integrated file system	1
What's new for V5R4	1
Printable PDF	1
Overview of the integrated file system.	2
What the integrated file system is	2
Why use the integrated file system	3
Integrated file system concepts	4
Directory	4
Current directory	7
Home directory	7
Provided directories	7
*TYPE2 directories	10
Link	11
Hard link	12
Symbolic link.	13
Path name.	14
Stream file.	16
Name continuity.	17
Extended attributes.	18
Scanning support	19
Examples: Scanning for viruses and files being opened	19
Related system values	20
Scanning occurrences	21
Object change	21
Signature change	22
Different CCSID	23
During save operation.	23
Object integrity check	23
File systems	24
File system comparison	25
"root" (/) file system	29
Case-sensitivity in the "root" (/) file system	29
Path names in the "root" (/) file system	30
Links in the "root" (/) file system	30
Integrated file system commands in the "root" (/) file system	30
Integrated file system APIs in the "root" (/) file system.	30
Object changes journaling in the "root" (/) file system	31
UDP and TCP devices in the "root" (/) file system	31
Open systems file system (QOpenSys)	31
Case-sensitivity in the QOpenSys file system	32
Path names in the QOpenSys file system	32
Links in the QOpenSys file system	32
Integrated file system commands and displays in the QOpenSys file system.	32
Integrated file system APIs in the QOpenSys file system.	33
Object changes journaling in the QOpenSys file system.	33
User-defined file systems (UDFSs).	33
Case-sensitivity in an integrated file system user-defined file system	34

Path names in an integrated file system user-defined file system	35
Links in an integrated file system user-defined file system.	36
Integrated file system commands in a user-defined file system	36
Integrated file system APIs in a user-defined file system.	37
Graphical user interface for a user-defined file system	37
Creating an integrated file system user-defined file system	37
Deleting an integrated file system user-defined file system.	37
Displaying an integrated file system user-defined file system	38
Mounting an integrated file system user-defined file system	38
Unmounting an integrated file system user-defined file system	38
Saving and restoring an integrated file system user-defined file system	38
Object changes journaling in a user-defined file system.	39
User-defined file system and independent auxiliary storage pools.	39
Library file system (QSYS.LIB)	39
QPWFSESERVER authorization list in the QSYS.LIB file system	40
File-handling restrictions in the QSYS.LIB file system	40
Support for user spaces in the QSYS.LIB file system	40
Support for save files in the QSYS.LIB file system	40
Case-sensitivity in the QSYS.LIB file system	41
Path names in the QSYS.LIB file system	41
Links in the QSYS.LIB file system	41
Integrated file system commands and displays in the QSYS.LIB file system	42
Integrated file system APIs in the QSYS.LIB file system.	42
Independent ASP QSYS.LIB	42
QPWFSESERVER authorization list in the independent ASP QSYS.LIB file system	43
File handling restrictions in the independent ASP QSYS.LIB file system	43
Support for user spaces in the independent ASP QSYS.LIB file system	43
Support for save files in the independent ASP QSYS.LIB file system	43
Case-sensitivity in the independent ASP QSYS.LIB file system	43
Path names in the independent ASP QSYS.LIB file system.	44

Links in the independent ASP QSYS.LIB file system	44	Links in the QFileSvr.400 file system	60
Integrated file system commands and displays in the independent ASP QSYS.LIB file system	44	Integrated file system commands and displays in the QFileSvr.400 file system	60
Integrated file system APIs in the independent ASP QSYS.LIB file system	45	Integrated file system APIs in the QFileSvr.400 file system.	60
Document library services file system (QDLS)	45	Network File System (NFS)	61
Integrated file system and HFS in the QDLS file system.	45	Characteristics of the Network File System	61
User enrollment in the QDLS file system	46	Variations of servers and clients in the Network File System	61
Case-sensitivity in the QDLS file system.	46	Links in the Network File System	62
Path names in the QDLS file system	46	Integrated file system commands in the Network File System	62
Links in the QDLS file system	46	Integrated file system APIs in the Network File System	63
Integrated file system commands and displays in the QDLS file system	47	Accessing the integrated file system	64
Integrated file system APIs in the QDLS file system	47	Accessing using menus and displays	64
Optical file system (QOPT)	48	Accessing using CL commands.	65
Integrated file system and HFS in the QOPT file system.	48	Path name rules for CL commands and displays	68
Case-sensitivity in the QOPT file system.	48	Working with output of the RTVDIRINF and PRTDIRINF commands	70
Path names in the QOPT file system	48	Accessing the data of RTVDIRINF.	81
Links in the QOPT file system	49	Using the data of RTVDIRINF	82
Integrated file system commands and displays in the QOPT file system	49	Accessing using APIs	82
Integrated file system APIs in the QOPT file system	49	Accessing using iSeries Navigator	82
NetWare file system (QNetWare)	50	Accessing using iSeries NetServer	83
Authorities and ownership in the QNetWare file system.	51	Accessing using File Transfer Protocol	84
Audition in the QNetWare file system	51	Accessing using a PC	85
Files and directories in the QNetWare file system	51	Converting directories from *TYPE1 to *TYPE2	85
NetWare Directory Services objects in the QNetWare file system	51	Overview of *TYPE1 to *TYPE2 conversion.	86
Links in the QNetWare file system.	51	Conversion considerations	86
Integrated file system commands and displays in the QNetWare file system.	51	Conversion status determination	86
Integrated file system APIs in the QNetWare file system.	52	User profiles creation	87
iSeries NetClient file system (QNTC)	53	Objects renamed.	87
Authorities and ownership in the QNTC file system	53	Combined characters	87
Case-sensitivity in the QNTC file system	53	Surrogate characters	88
Path names in the QNTC file system	53	User profile considerations	88
Links in the QNTC file system	54	Changing maximum storage for a user profile	88
Integrated file system commands and displays in the QNTC file system	54	Changing the owner of a directory	88
QNTC environment variables	55	Auxiliary storage requirements	88
Creating directories in the QNTC file system	55	Tips: Symbolic link	89
Integrated file system APIs in the QNTC file system	56	Tips: Independent ASP	89
Enabling QNTC file system for Network Authentication Service.	56	Tips: Saving and restoring	89
i5/OS file server file system (QFileSvr.400)	57	Tips: Reclaiming integrated file system objects	90
Case-sensitivity in the QFileSvr.400 file system	57	Integrated file system scanning.	90
Path names in the QFileSvr.400 file system	58	Journaling objects	90
Communications in the QFileSvr.400 file system	58	Journaling overview	90
Security and object authority in the QFileSvr.400 file system	59	Journal management	91
		Objects you should journal	91
		Journalled integrated file system objects	91
		Journalled operations	93
		Special considerations for journal entries	93
		Considerations for multiple hard links and journaling	94
		Starting journaling	95
		Changing journaling	95
		Ending journaling	95
		Reclaim operation of the "root" (/), QOpenSys, and user-defined file systems	96

Reclaim Object Links (RCLLNK) and Reclaim Storage (RCLSTG) commands comparison	96	Naming and international support	113
Reclaim Object Links (RCLLNK) command	97	Data conversion	114
Re-creation of integrated file system provided objects	98	Example: Integrated file system C functions	115
Examples: Reclaim Object Links (RCLLNK) command	98	Working with files and folders using iSeries Navigator	120
Example: Correcting problems for an object	98	Checking in a file	120
Example: Correcting problems that exist in a directory subtree	98	Checking out a file	120
Example: Finding all damaged objects in the "root" (/), QOpenSys, and mounted user-defined file systems	99	Creating a folder	120
Example: Deleting all damaged objects in the "root" (/), QOpenSys, and mounted user-defined file systems	99	Removing a folder.	120
Example: Running multiple RCLLNK commands to quickly reclaim all objects in the "root" (/), QOpenSys, and mounted user-defined file systems	99	Moving files or folders to another file system	121
Programming support	100	Setting permissions	122
Copying data between stream files and database files	100	Setting up file text conversion	122
Copying data using CL commands	100	Sending a file or folder to another system	123
Copying data using APIs	101	Changing options for a package definition.	123
Copying data using data-transfer functions	101	Scheduling a date and time to send your file or folder	124
Transferring data from a database file to a stream file	102	Creating a file share	124
Transferring data from a stream file to a database file.	102	Changing a file share.	124
Transferring data into a newly created database file definition and file	103	Creating a new user-defined file system	124
Creating a format description file.	103	Mounting a user-defined file system.	124
Copying data between stream files and save files	104	Unmounting a user-defined file system.	125
Performing operations using APIs	104	Setting whether objects should be scanned or not	125
ILE C functions.	109	Transport-independent remote procedure call.	126
Large file support	110	Network selection APIs	126
Path name rules for APIs	111	Name-to-address translation APIs	127
File descriptor	112	eXternal Data Representation (XDR) APIs	127
Security	112	Authentication APIs	129
Socket support	113	Transport-independent RPC (TI-RPC) APIs	129
		TI-RPC simplified APIs	129
		TI-RPC top-level APIs	129
		TI-RPC intermediate-level APIs	130
		TI-RPC expert-level APIs	130
		Other TI-RPC APIs	130
		Related information for integrated file system	131
		Appendix. Notices	133
		Programming Interface Information	134
		Trademarks	135
		Terms and conditions.	135

Integrated file system

The integrated file system is a part of the i5/OS[®] operating system that supports stream input/output and storage management similar to personal computer and UNIX[®] operating systems, while providing you with an integrating structure over all information stored in the system.

Note: By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 132.

What’s new for V5R4

This topic highlights the changes made to this topic collection for V5R4.

QNTC

iSeries[®] NetClient (QNTC) file system now supports TCP/IP port 445 and larger file sizes.

- QNTC supports TCP/IP port 445

The QNTC file system is now capable of contacting systems using TCP/IP port 445. The Windows[®] servers do not need to be configured for NetBios over TCP/IP.

- QNTC supports large file sizes

Beginning with V5R4, the QNTC file system supports reading and writing files up to a size of 1 TB (1 TB equals approximately 1 099 511 627 776 bytes).



Reclaim Object Links (RCLLNK) command

The Reclaim Object Links (RCLLNK) command identifies and repairs damaged objects in the “root” (/), QOpenSys, and mounted user-defined file systems without requiring the system to be in a restricted state. This enables you to correct problems in these file systems without sacrificing productivity. It can be used as an alternative to the Reclaim Storage (RCLSTG) command in many situations. For example, RCLLNK is ideal for identifying and correcting problems in the following situations:

- Problems are isolated to a single object.
- Problems are isolated to a group of objects.
- Damaged objects need to be identified or deleted.
- The system cannot be in a restricted state during the reclaim operation.
- Independent auxiliary storage pools (ASPs) must be available during the reclaim operation.

How to see what’s new or changed

To help you see where technical changes have been made, this information uses:

- The  image to mark where new or changed information begins.
- The  image to mark where new or changed information ends.

To find other information about what’s new or changed this release, see the Memo to users.

Printable PDF

Use this to view and print a PDF of this information.


To view or download the PDF version of this document, select Integrated file system (about 1845 KB).

Saving PDF files

To save a PDF on your workstation for viewing or printing:

1. Right-click the PDF in your browser (right-click the link above).
2. Click the option that saves the PDF locally.
3. Navigate to the directory in which you want to save the PDF.
4. Click **Save**.

Downloading Adobe Reader

- l You need Adobe Reader installed on your system to view or print these PDFs. You can download a free copy from the Adobe Web site (www.adobe.com/products/acrobat/readstep.html) .

Overview of the integrated file system

Here is some basic information about the integrated file system on your i5/OS operating system and how it can be of use on your system.

What the integrated file system is

The *integrated file system* is a part of the i5/OS operating system that supports stream input/output and storage management similar to personal computer and UNIX operating systems, while providing an integrating structure over all information stored on your system.

The integrated file system comprises 11 file systems, each with their own set of logical structures and rules for interacting with information in storage.

The key features of the integrated file system are as follows:

- Support for storing information in stream files that can contain long continuous strings of data. These strings of data might be, for example, the text of a document or the picture elements in a picture. The stream file support is designed for efficient use in client/server applications.
- A hierarchical directory structure that allows objects to be organized like fruit on the branches of a tree. You can access an object by specifying the path through the directory to the object.
- A common interface that enables users and applications to access not only the stream files but also database files, documents, and other objects that are stored on your system.
- A common view of stream files that are stored locally on your system, Integrated xSeries® Server for iSeries, or a remote Windows NT® server. Stream files can also be stored remotely on a local area network (LAN) server, a Novell NetWare server, another remote System i™ product, or a Network File System (NFS) server.

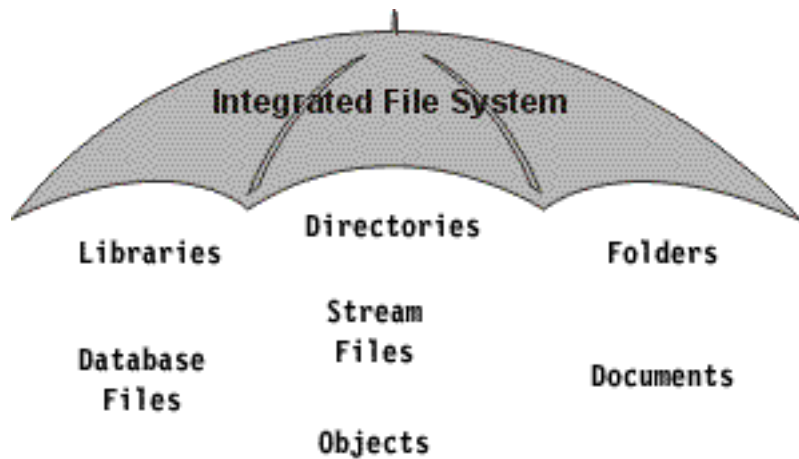


Figure 1. A structure over all information stored in the i5/OS operating system

Related concepts

“File systems” on page 24

A *file system* provides you with the support to access specific segments of storage that are organized as logical units. These logical units on your system are files, directories, libraries, and objects.

Why use the integrated file system

The integrated file system enhances the already extensive data management capabilities of i5/OS with additional capabilities to better support emerging and future forms of information processing, such as client/server, open systems, and multimedia.

You can use the integrated file system to do the following things:

- Provide fast access to i5/OS data, especially for applications such as iSeries Access that use the i5/OS file server.
- Allow more efficient handling of types of stream data, such as images, audio, and video.
- Provide a file system base and a directory base for supporting open system standards based on the UNIX operating system, such as Portable Operating System Interface for Computer Environments (POSIX) and X/Open Portability Guide (XPG). This file structure and this directory structure also provides a familiar environment for users of PC operating systems, such as Disk Operating System (DOS) and Windows operating systems.
- Allow file support with unique capabilities (such as record-oriented database files, UNIX operating system-based stream files, and file serving) to be handled as separate file systems, while allowing them all to be managed through a common interface.

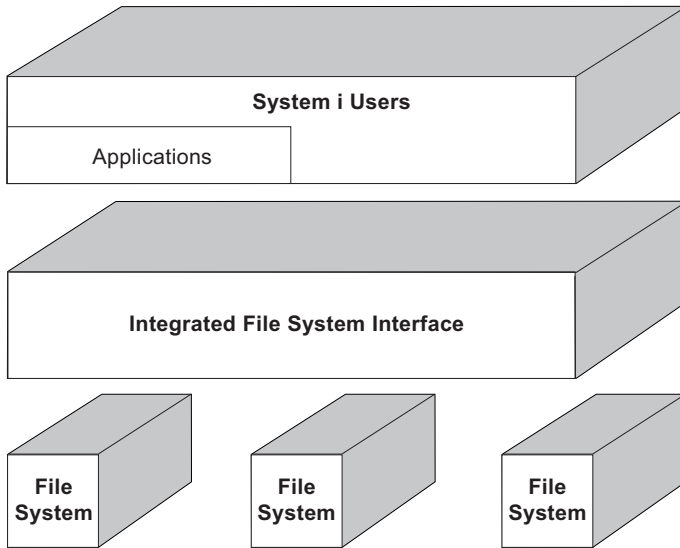


Figure 2. A common interface to separate file systems

- Allow PC users to take better advantage of their graphical user interface. For example, Windows users can use the Windows graphical tools to operate on i5/OS stream files and other objects in the same way they operate on files stored on their PCs.
- Provide continuity of object names and associated object information across national languages. For example, this ensures that individual characters remain the same when switching from the code page of one language to the code page of another language.

Related concepts

“File systems” on page 24

A *file system* provides you with the support to access specific segments of storage that are organized as logical units. These logical units on your system are files, directories, libraries, and objects.

Integrated file system concepts

This topic introduces the basic concepts of integrated file system, such as directory, link, path name, stream file, name continuity, extended attributes, and scanning support.

Directory

A *directory* is a special object that is used to locate objects by names that you specify. Each directory contains a list of objects that are attached to it. That list can include other directories.

The integrated file system provides a hierarchical directory structure for you to access all objects in your system. You might think of this directory structure as an inverse tree where the root is at the top and the branches below. The branches represent directories in the directory hierarchy. These directory branches have subordinate branches that are called subdirectories. Attached to the various directory and subdirectory branches are objects such as files. Locating an object requires specifying a path through the directories to the subdirectory to which the object is attached. Objects that are attached to a particular directory are sometimes described as being *in* that directory.

A particular directory branch, along with all of its subordinate branches (subdirectories) and all of the objects that are attached to those branches, is referred to as a *subtree*. Each file system is a major subtree in the integrated file system directory structure. In the QSYS.LIB and independent ASP QSYS.LIB file systems’ subtrees, a library is handled the same way as a subdirectory. Objects in a library are handled like objects in a subdirectory. Because database files contain objects (database file members), they are

handled like subdirectories rather than objects. In the document library services file system (QDLS subtree), folders are handled like subdirectories and documents in folders are handled like objects in a subdirectory.

Because of differences in file systems, the operations you can perform in one subtree of the directory hierarchy may not work in another subtree.

The integrated file system directory support is similar to the directory support that is provided by the DOS file system. In addition, it provides features typical of UNIX systems, such as the ability to store a file only once but access it through multiple paths by using links.

File systems and objects are branches on the integrated file system directory tree. See the following figure for an example of an integrated file system directory tree.

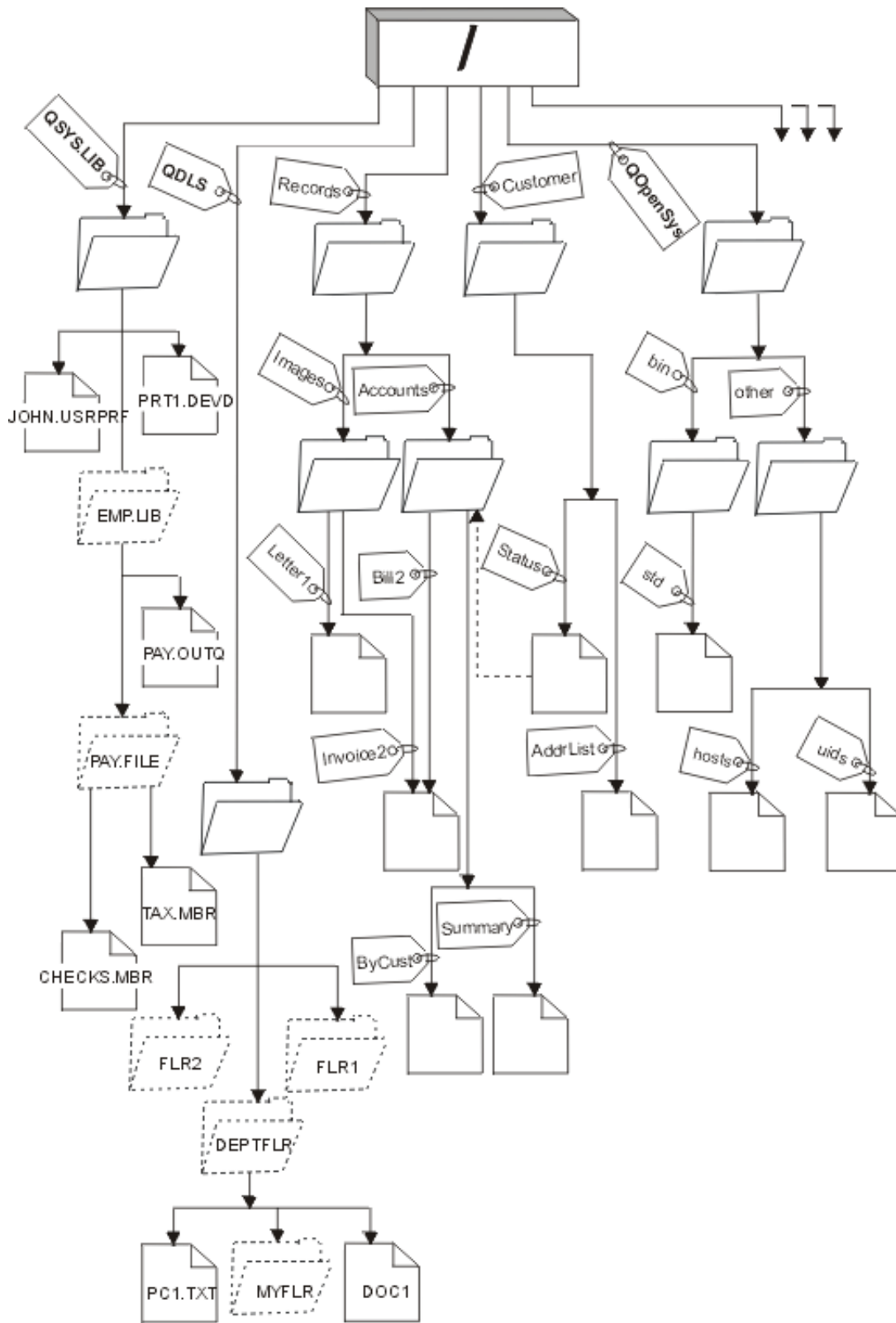


Figure 3. Sample integrated file system directory tree

Current directory

The current directory is similar to the idea of the current library. It is also called the *current working directory*, or just *working directory*.

Your *current directory* is the first directory in which the operating system looks for your programs and files and stores your temporary files and output. When you request an operation on an object, such as a file, the system searches for the object in your current directory unless you specify a different directory path.

Home directory

The *home directory* is used as the current directory when you sign on to the system. The name of the home directory is specified in your user profile.

When your job is started, the system looks in your user profile for the name of your home directory. If a directory by that name does not exist on the system, the home directory is changed to the "root" (/) directory.

Typically, the system administrator who creates the user profile for a user also creates the user's home directory. Creating individual home directories for each user under the /home directory is preferable. The /home directory is a subdirectory under the "root" (/) directory. The system default expects the name of the home directory of a user to be the same name as the user profile.

For example, the command `CRTUSRPRF USRPRF(John) HOMEDIR(*USRPRF)` will assign the home directory for John to /home/JOHN. If the directory /home/JOHN does not exist, the "root" (/) directory becomes the home directory for John.

You can specify a directory other than the home directory as your current directory at any time after you sign on by using the Change Current Directory (CHGCURDIR) CL command, the `chdir()` API, or the `fchdir()` API.

The home directory chosen during process initiation will remain each thread's home directory by default. This is true regardless of whether your active user profile for the thread has changed after initiation. However, there is support provided by the Change Job (QWTCHGJB) API that can be used to change the home directory being used for a thread to that thread's current user profile's home directory (or the "root" (/) directory if that home directory does not exist). Secondary threads will always inherit the home directory of the thread that created it. Note that the process' current directory does not change when you use QWTCHGJB to change the thread's home directory. The current directory is scoped to the process level, and the home directory is scoped to the thread level. Changing the current working directory in any thread changes it for the whole process. Changing the home directory for a thread does not change its current working directory.

Related information

Change current directory (CHGCURDIR) command

`chdir()`--Change Current Directory API

`fchdir()`--Change Current Directory by Descriptor API

Application programming interfaces (APIs)

Provided directories

The integrated file system creates these directories when the system is restarted if they do not already exist.

Note: Do not replace the following system-created directories with symbolic links to other objects. For example, do not replace /home with a symbolic link to a directory on an independent ASP. Otherwise, there might be problems on the independent ASP as well as problems when creating new user profiles.

/tmp The /tmp directory gives applications a place to store temporary objects. This directory is a subdirectory of the "root" (/) directory, so its path name is /tmp.

Once an application puts an object in the /tmp directory, the object stays there until you or the application removes it. The system does not automatically remove objects from the /tmp directory or perform any other special processing for objects in the /tmp directory.

You can use the user displays and commands that support the integrated file system to manage the /tmp directory and its objects. For example, you can use the Work with Object Links display or the WRKLNK command to copy, remove, or rename the /tmp directory or objects in the directory. All users are given *ALL authority to the directory, which means that they can perform most valid actions on the directory.

An application can use the application programming interfaces (API) that support the integrated file system to manage the /tmp directory and its objects. For example, the application program can remove an object in the /tmp directory by using the unlink() API.

If the /tmp directory is removed, it is automatically created again during the next restart of the system.

The /tmp directory can have the restricted rename and unlink attribute set to Yes for operating system commonality and security purposes.

Note: The restricted rename and unlink attribute is equivalent to the S_ISVTX mode bit for a directory.

If the restricted rename and unlink attribute is set to Yes, you cannot rename or unlink objects within the /tmp directory unless one of the following conditions is true:

- You are the owner of the object.
- You are the owner of the directory.
- You have all object (*ALLOBJ) special authority.

If the attribute is set to Yes, and you do not have the appropriate authorities, you will see error number 3027 (EPERM) or message MSGCPFA0B1 (Requested operation not allowed. Access problem) for rename or unlink failures when using the following commands and APIs:

- Remove Link (RMVLNK, DEL, and ERASE) command
- Remove Directory (RMVDIR, RD, and RMDIR) command
- Rename Object (RNM and REN) command
- Move Object (MOV and MOVE) command
- Rename File or Directory (rename()) API
- Rename File or Directory, Keep "new" If It Exists (Qp0lRenameKeep()) API
- Rename File or Directory, Unlink "new" If It Exists (Qp0lRenameUnlink()) API
- Remove Directory (rmdir()) API
- Remove Link to File (unlink()) API

The restricted rename and unlink attribute and S_ISVTX mode bit can be modified using the Change Attribute (CHGATR) command or the Set Attributes (Qp0lSetAttr()) or Change File Authorizations (chmod) APIs if you are the owner of the object, or if you have all object (*ALLOBJ) special authority. But, if the attribute is changed to No, you will lose the operating system commonality and security benefits that the Yes setting provides.

When the /tmp directory is created during a restart of the system, the attribute is set to Yes. If the /tmp directory already exists during a restart of the system, the attribute is not changed.

/home System administrators use the /home directory to store a separate directory for every user. The system administrator often sets the home directory that is associated with the user profile to be the user's directory in /home, for example/home/john.

- /etc** The /etc directory stores administrative, configuration, and other system files.
- /usr** The /usr directory includes subdirectories that contain information that is used by the system. Files in /usr typically do not change often.
- /usr/bin**
The /usr/bin directory contains the standard utility programs.
- /QIBM**
The /QIBM directory is the system directory and is provided with the system.
- /QIBM/ProdData**
The /QIBM/ProdData directory is a system directory used for Licensed program data.
- /QIBM/UserData**
The /QIBM/UserData directory is a system directory used for Licensed Program user data such as configuration files.
- /QOpenSys/QIBM**
The /QOpenSys/QIBM directory is the system directory for the QOpenSys file system.
- /QOpenSys/QIBM/ProdData**
The /QOpenSys/QIBM/ProdData directory is the system directory for the QOpenSys file system and is used for Licensed program data.
- /QOpenSys/QIBM/UserData**
The /QOpenSys/QIBM/UserData directory is the system directory for the QOpenSys file system and is used for Licensed Program user data such as configuration files.
- /asp_name/QIBM**
The /asp_name/QIBM directory is the system directory for any independent ASPs that exist on your system, where asp_name is the name of the independent ASP.
- /asp_name/QIBM/UserData**
The /asp_name/QIBM/UserData directory is a system directory used for Licensed Program user data such as configuration files for any independent ASPs that exist on your system, where asp_name is the name of the independent ASP.
- /dev** The /dev directory contains various system files and directories.
- /dev/xti**
The /dev/xti directory contains the UDP and TCP device drivers.

Related concepts

“Home directory” on page 7

The *home directory* is used as the current directory when you sign on to the system. The name of the home directory is specified in your user profile.

Related reference

“UDP and TCP devices in the “root” (/) file system” on page 31

The “root” (/) file system under the directory of /dev/xti will now hold two device drivers named udp and tcp.

“Open systems file system (QOpenSys)” on page 31

The QOpenSys file system is compatible with open system standards based on UNIX, such as POSIX and X/Open Portability Guide (XPG). Like the “root” (/) file system, this file system takes advantage of the stream file and directory support that is provided by the integrated file system.

Related information

Work with Object Links (WRKLNK) command

***TYPE2 directories**

The "root" (/), QOpenSys, and user-defined file systems (UDFS) in the integrated file system support the *TYPE2 directory format. The *TYPE2 directory format is an enhancement of the original *TYPE1 directory format.

Note: The concept of *TYPE1 and *TYPE2 stream files is different from the concept of *TYPE1 and *TYPE2 directory formats. One does not relate to the other.

*TYPE2 directories have a different internal structure and different implementation than *TYPE1 directories.

The advantages of *TYPE2 directories are:

- Improved performance
- Improved reliability
- Added functionality
- In many cases, less auxiliary storage space

*TYPE2 directories improve file system performance over *TYPE1 directories, especially when creating and deleting directories.

*TYPE2 directories are more reliable than *TYPE1 directories. After a system abnormally ends, *TYPE2 directories are completely recovered unless there has been an auxiliary storage failure. *TYPE1 directories may require the use of the Reclaim Storage (RCLSTG) command in order to recover completely.

*TYPE2 directories provide the following added functionality:

- *TYPE2 directories support renaming the case of a name in a monospace file system (for example, renaming from A to a).
- An object in a *TYPE2 directory can have up to one million links compared to 32 767 links for *TYPE1 directories. This means you can have up to 1 million hard links to a stream file, and a *TYPE2 directory can contain up to 999 998 subdirectories.
- Using iSeries Navigator, the list of entries are automatically sorted in binary order when you open a directory that has the *TYPE2 format.
- Some new functions such as integrated file system scanning support are only available for objects in *TYPE2 directories.

Typically, *TYPE2 directories that have fewer than 350 objects require less auxiliary storage than *TYPE1 directories with the same number of objects. *TYPE2 directories with more than 350 objects are ten percent larger (on average) than *TYPE1 directories.

There are several ways to get *TYPE2 directories on your system:

- New System i platforms that are preinstalled with OS/400® V5R2 or i5/OS V5R3, or later, have *TYPE2 directories. No conversion is needed for "root" (/), QOpenSys, and UDFSs in ASPs 1-32.
- If you install OS/400 V5R2 or i5/OS V5R3, or later, on a System i platform for the first time, the platform has *TYPE2 directories. No conversion is needed for "root" (/), QOpenSys, and UDFSs in ASPs 1-32.
- The V5R1 or V5R2 conversion utility is used to convert the file systems.
- If the UDFSs in an independent ASP have not yet been converted to the *TYPE2 format, they will be converted the first time that the independent ASP is varied on to a system installed with OS/400 V5R2 or i5/OS V5R3 or later.
- All other supported file systems except UDFSs on independent ASPs that are still using *TYPE1 directories are converted automatically by the system. This conversion begins after the installation of i5/OS V5R3 or later releases. It should not significantly impact your system activity.

To determine the directory format for the file systems on your system, use the Convert Directory (CVTDIR) command:

```
CVTDIR OPTION(*CHECK)
```

Note: *TYPE2 directories are supported on OS/400 V5R2 or i5/OS V5R3 or later, but there are some differences from normal *TYPE2 directory support.

***TYPE2 directories in OS/400 V5R1 or V5R2**

The "root" (/), QOpenSys, and user-defined file systems (UDFS) in the integrated file system support the *TYPE2 directory format in OS/400 V5R1, V5R2, and later.

The *TYPE2 directory format is an enhancement of the original *TYPE1 directory format. *TYPE2 directories have a different internal structure from *TYPE1 directories and provide improved performance and reliability.


If you have OS/400 V5R1 or V5R2, you can convert your directories to the *TYPE2 directory format using the appropriate conversion utility. Shortly after i5/OS V5R3 or a later release is installed, the conversion to *TYPE2 directories will automatically begin for any of the file systems that have not yet been converted to support *TYPE2 directories. Therefore, you might want to consider converting to the *TYPE2 directory format before installing a later version to avoid this automatic conversion.

The support for *TYPE2 directories in OS/400 V5R2 is available in the V5R2 iSeries Information Center through the Convert Directory (CVTDIR) command.

The support for *TYPE2 directories in OS/400 V5R1 is available through program temporary fixes (PTFs). The conversion utility is slightly different from the OS/400 V5R2 version. Refer to the informational APAR II13161 for complete documentation on *TYPE2 directories in V5R1. Use one of the following methods to access the APAR:

- Download the informational APAR to your system and view it. Use the following commands:

```
SNDPTFORD PTFID((II13161))  
DSPPTFCVR LICPGM(INFOAS4) SELECT(II13161)
```

- Go to the Support for IBM System i  Web site to view the informational APAR. See **Problem Solving** → **Technical Databases** → **Authorized Program Analysis Reports (APARs)** → **V5R1 APARs** → **INFOAS400 - AS/400 Information** → **APAR number II13161**.

Related reference

"Converting directories from *TYPE1 to *TYPE2" on page 85

The "root" (/), QOpenSys, and user-defined file systems (UDFS) in the integrated file system support the *TYPE2 directory format as of OS/400 V5R1.

Related information

Reclaim Storage (RCLSTG) command

Convert Directory (CVTDIR) command

Link

A *link* is a named connection between a directory and an object. A user or a program can tell the system where to find an object by specifying the name of a link to the object. A link can be used as a path name or as part of a path name.

For users of directory-based file systems, it is convenient to think of an object, such as a file, as something that has a name that identifies it to the system. In fact, it is the directory path to the object that identifies it. You can sometimes access an object by giving just the object's *name*. You can do this because

the system is designed to assume the directory part of the path under certain conditions. The idea of a link takes advantage of the reality that it is the directory path that identifies the object. The name is given to the link rather than the object.

After you get used to the idea that the link has the name rather than the object, you begin to see possibilities that were hidden before. There can be multiple links to the same object. For example, two users can share a file by having a link from each user's home directory to the file (see "Home directory" on page 7). Certain types of links can cross file systems, and can exist without an object existing.

There are two types of links: Hard link and Symbolic link. When using path names in programs, you have a choice of using a hard link or a symbolic link. Each type of link has advantages and disadvantages. The conditions under which one type of link has an advantage over the other type is as follows:

Table 1. Comparison of hard link and symbolic link

Item	Hard link	Symbolic link
Name resolution	Faster. A hard link contains a direct reference to the object.	Slower. A symbolic link contains a path name to the object, which must be resolved to find the object.
Object existence	Required. An object must exist in order to create a hard link to it.	Optional. A symbolic link can be created when the object it refers to does not exist.
Object deletion	Restricted. All hard links to an object must be unlinked (removed) to delete the object.	Unrestricted. An object can be deleted even if there are symbolic links referring to it.
Static objects (attributes do not change)	Faster. For a static object, name resolution is the primary performance concern. Name resolution is faster when hard links are used.	Slower. Name resolution is slower when symbolic links are used.
Scope	Restricted. Hard links cannot cross file systems.	Unrestricted. Symbolic links can cross file systems.

Hard link

A *hard link*, which is sometimes just called a link, cannot exist unless it is linked to an actual object.

When an object is created in a directory (for example, by copying a file into a directory), the first hard link is established between the directory and the object. Users and application programs can add other hard links. Each hard link is indicated by a separate directory entry in the directory. Links from the same directory cannot have the same name, but links from different directories can have the same name.

If supported by the file system, there can be multiple hard links to an object, either from the same directory or from different directories. The one exception is where the object is another directory. There can be only one hard link from a directory to another directory.

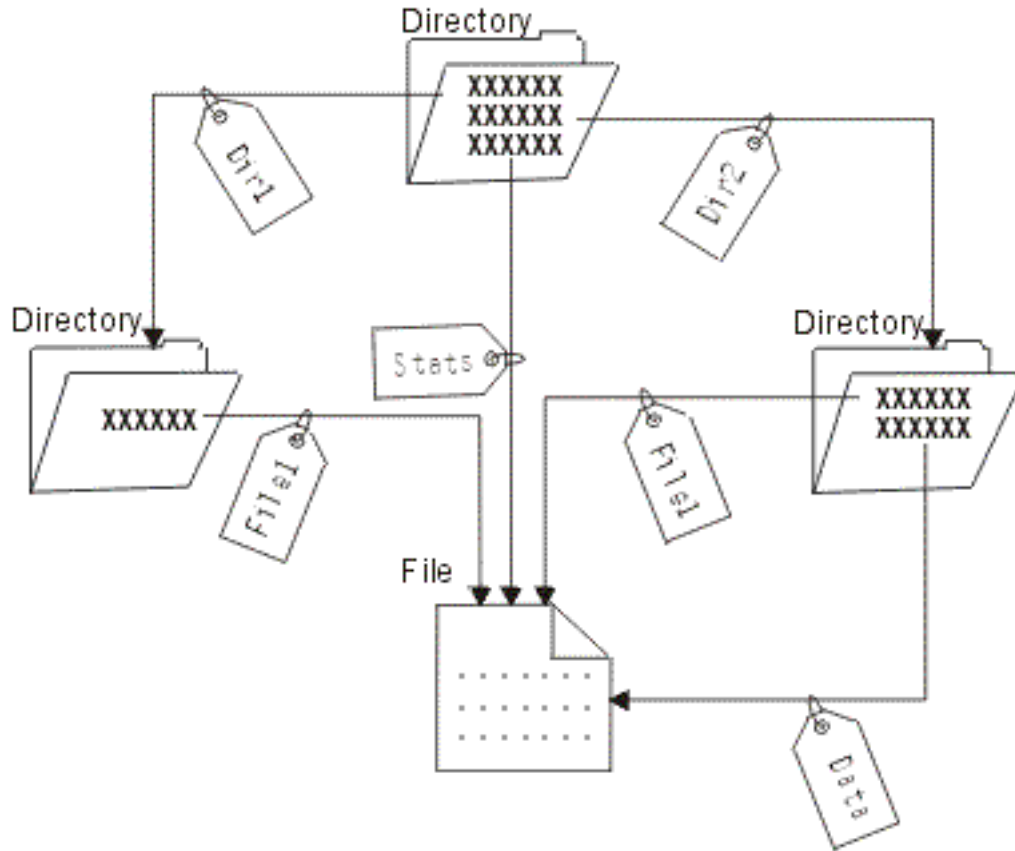


Figure 4. A directory entry defines each hard link

Hard links can be removed without affecting the existence of an object as long as there is at least one remaining hard link to the object. When the last hard link is removed, the object is removed from the system, unless an application has the object open. Each application that has the object open can continue to use it until that application closes the object. When the object is closed by the last application using it, the object is removed from the system. An object cannot be opened after the last hard link is removed.

The concept of a hard link can also be applied to the QSYS.LIB or independent ASP QSYS.LIB file systems and the document library services (QDLS) file system, but with a restriction. A library, in effect, has one hard link to each object in the library. Similarly, a folder has one hard link to each document in the folder. Multiple hard links to the *same object* are not allowed in QSYS.LIB, independent ASP QSYS.LIB, or in QDLS, however.

A hard link cannot cross file systems. For example, a directory in the QOpenSys file system cannot have a hard link to an object in the QSYS.LIB or independent ASP QSYS.LIB file systems or to a document in the QDLS file system.

Symbolic link

A *symbolic link*, which is also called a soft link, is a path name contained in a file.

When the system encounters a symbolic link, it follows the path name provided by the symbolic link and then continues on any remaining path that follows the symbolic link. If the path name begins with a /, the system returns to the / ("root") directory and begins following the path from that point. If the path name does not begin with a /, the system returns to the immediately preceding directory and follows the path name in the symbolic link beginning at that directory.

Consider the following example of how a symbolic link might be used:

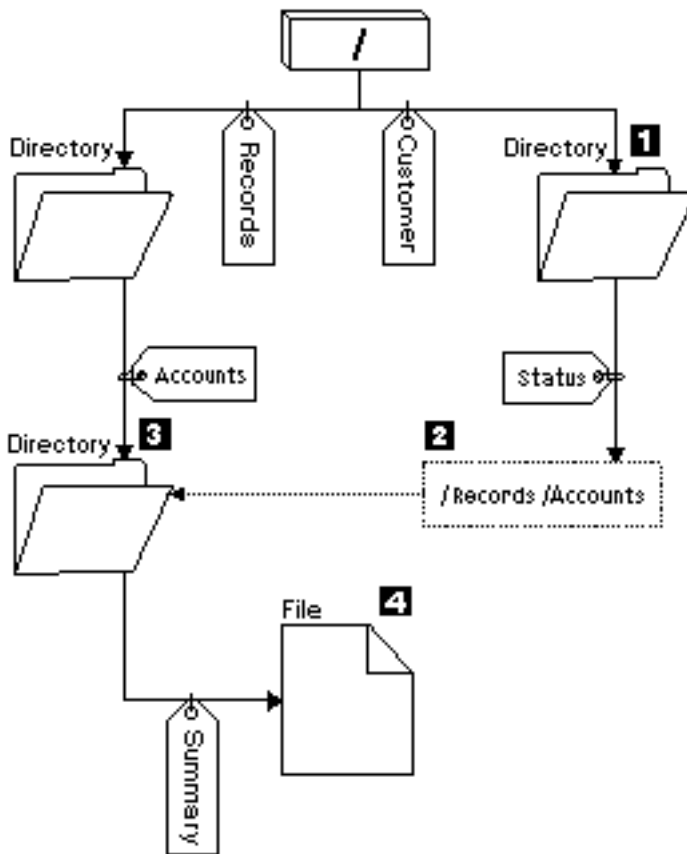


Figure 5. An example of using a symbolic link

You select a menu option to show the status of customer accounts. The program displaying the menu uses the following path name:

```
/Customer/Status/Summary
```

The system follows the *Customer* link, which leads to a directory 1, and then follows the *Status* link. The *Status* link is a symbolic link, which contains a path name 2. Because the path name begins with a /, the system returns to the / ("root") directory and follows the links *Records* and *Accounts* in sequence. This path leads to another directory 3. Now the system completes the path in the path name provided by the program. It follows the *Summary* link, which leads to a file 4 containing the data you will need.

Unlike a hard link, a symbolic link is an object (of object type *SYMLNK); it can exist without pointing to an object that exists. You can use a symbolic link, for example, to provide a path to a file that will be added or replaced later.

Also unlike a hard link, a symbolic link can cross file systems. For example, if you are working in one file system, you can use a symbolic link to access a file in another file system. Although the QSYS.LIB, independent ASP QSYS.LIB, and QDLS file systems do not support creating and storing symbolic links, you can create a symbolic link in the "root" (/) or QOpenSys file system that allows you to:

- Access a database file member in the QSYS.LIB or independent ASP QSYS.LIB file systems.
- Access a document in the QDLS file system.

Path name

A *path name* (also called a *pathname* on some systems) tells the system how to locate an object.

The path name is expressed as a sequence of directory names followed by the name of the object. Individual directories and the object name are separated by a slash (/) character; for example:

```
directory1/directory2/file
```

For your convenience, the backslash (\) can be used instead of the slash in integrated file system commands.

There are two ways of indicating a path name:

- An *absolute path name* begins at the highest level, or “root” directory (which is identified by the / character). For example, consider the following path from the / directory to the file named Smith.

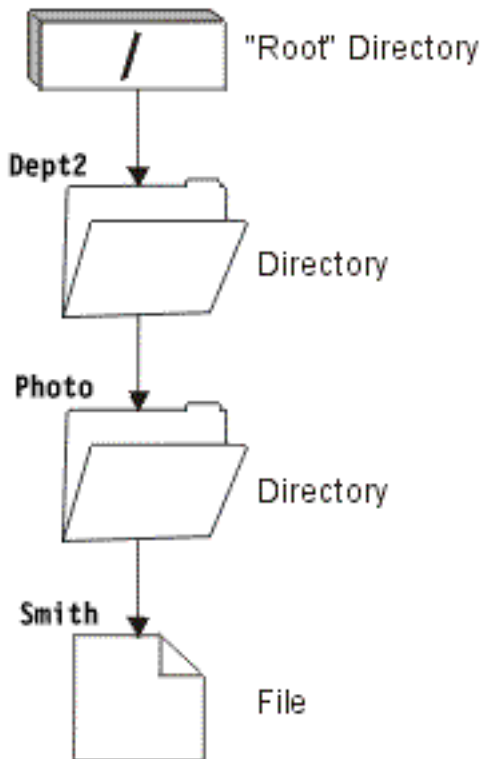


Figure 6. The components of a path name

The absolute path name to the Smith file is as follows:

```
/Dept2/Photo/Smith
```

The absolute path name is also known as the *full path name*.

- If the path name does not begin with the / character, the system assumes that the path begins at your current directory. This type of path name is called a *relative path name*. For example, if your current directory is Dept2 and it has a subdirectory named Photo containing the file Smith, the relative path name to the file is:

```
Photo/Smith
```

Notice that the path name does not include the name of the current directory. The first item in the name is the directory or object at the *next level below* the current directory.

Related reference

“Path name rules for APIs” on page 111

When using an integrated file system or ILE C API to operate on an object, you identify the object by supplying its directory path. Here is a summary of rules to keep in mind when specifying path names in the APIs.

“Path name rules for CL commands and displays” on page 68

When using an integrated file system command or display to operate on an object, you identify the object by supplying its path name.

Stream file

A *stream file* is a randomly accessible sequence of bytes, with no further structure imposed by the system.

The integrated file system provides support for storing and operating on information in the form of stream files. Documents that are stored in your system’s folders are stream files. Other examples of stream files are PC files and the files in UNIX systems. An integrated file system stream file is a system object that has an object type of *STMF.

To better understand stream files, it is useful to compare them with i5/OS database files. A database file is record-oriented; it has predefined subdivisions that consist of one or more fields that have specific characteristics, such as length and data type.

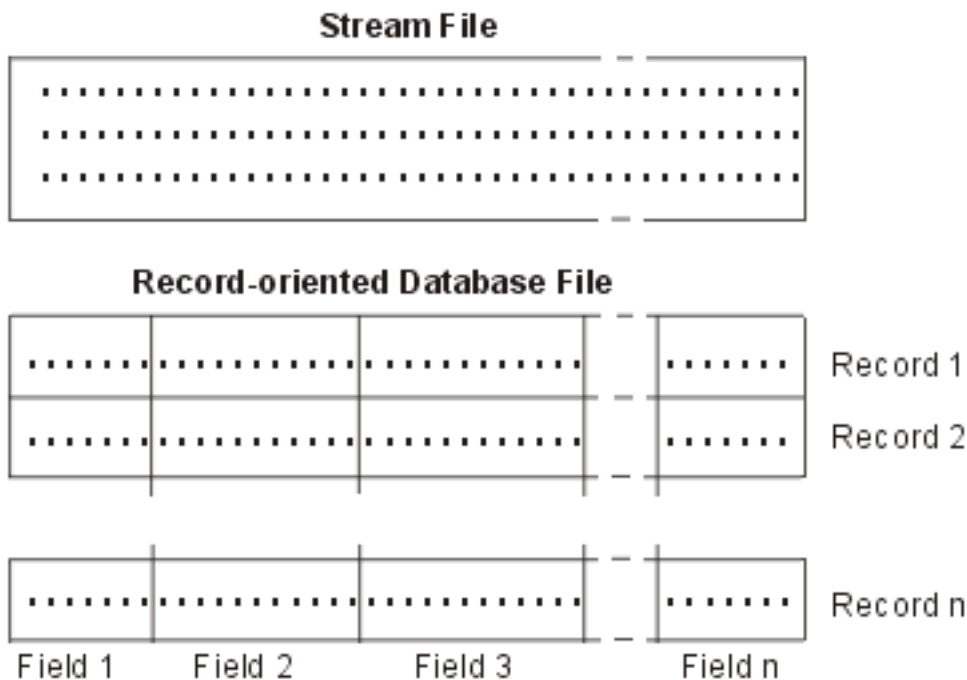


Figure 7. Comparison of a stream file and a record-oriented file

Stream files and record-oriented files are structured differently, and this difference in structure affects how the files are used. The structure affects how an application is written to interact with the files and where each type of file is best used in an application. A record-oriented file, for example, is well suited for storing customer statistics such as name, address, and account balance. A record-oriented file allows these predefined fields to be individually accessed and manipulated, using the extensive programming facilities of your system. But a stream file is better suited for storing information, such as a customer’s picture, which is composed of a continuous string of bits representing variations in color. Stream files are particularly well suited for storing strings of data, such as the text of a document, images, audio, and video.

A file has one of two format options: *TYPE1 stream file or *TYPE2 stream file. The file format depends on the release the file was created on, or if a file was created in a user-defined file system, the value that was specified for that file system.

Note: The concept of *TYPE1 and *TYPE2 stream files is different from the concept of *TYPE1 and *TYPE2 directory formats. One does not relate to the other.

*TYPE1 stream files

A *TYPE1 stream file has the same format as stream files created on releases before OS/400 V4R4.

- | A *TYPE1 stream file has a minimum size of 4096 bytes and a maximum object size of approximately 128 GB (1 GB equals approximately 1 073 741 824 bytes).

*TYPE2 stream files

A *TYPE2 stream file has high-performance file access.

*TYPE2 stream files have a maximum object size of approximately 1 TB (1 TB equals approximately 1 099 511 627 776 bytes) in the "root" (/), QOpenSys and user-defined file systems. Otherwise, the maximum is approximately 256 GB. It is also capable of memory mapping, as well as the ability to specify an attribute to optimize main storage allocation. All files created with OS/400 V4R4 and newer systems are *TYPE2 stream files, unless they were created in a user-defined file system that specified a file format of *TYPE1.

Note: Any files larger than 256 GB cannot be saved or restored to systems before i5/OS V5R3.

Name continuity

When you use the "root" (/), QOpenSys, and user-defined file systems, you can take advantage of system support that ensures characters in object names remain the same.

This also applies when you use these file systems across systems and connected devices that have different character encoding schemes (code pages). Your system stores the characters in the names in a 16-bit form that is known as UCS2 Level 1 (also called *Unicode*) for *TYPE1 directories and UTF-16 for *TYPE2 directories. UCS2 Level 1 and UTF-16 are subsets of the ISO 10646 standard. When the name is used, the system converts the stored form of the characters into the appropriate character representation in the code page being used. The names of extended attributes associated with each object are also handled the same way.

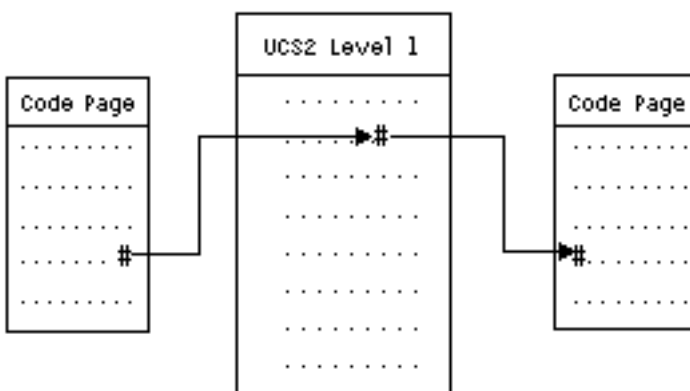


Figure 8. Keep characters the same across encoding schemes

This support makes it easier to interact with a system from devices using different code pages. For example, PC users can access an i5/OS file using the same file name, even though their PCs do not have the same code page as your system. The conversion from one code page to another is handled automatically by your system. Of course, the device must be using a code page that contains the characters used in the name.

Related concepts

“*TYPE2 directories” on page 10

The “root” (/), QOpenSys, and user-defined file systems (UDFS) in the integrated file system support the *TYPE2 directory format. The *TYPE2 directory format is an enhancement of the original *TYPE1 directory format.

Extended attributes

An extended attribute is information associated with an object that provides additional details about the object. The extended attribute consists of a name, which is used to refer to it, and a value. The value can be text, binary data, or another type of data.

The extended attributes for an object exist only as long as the object exists.

Extended attributes come in many varieties and can be used to contain a variety of information. You may need to be aware of the following three extended attributes, in particular:

.SUBJECT

A brief description of the content or purpose of the object.

.TYPE The type of data in the object. The type of data might be text, binary, source for a program, a compiled program, or other information.

.CODEPAGE

The code page to be used for the object. The code page used for the object is also used for the extended attribute associated with the object.

A period (.) as the first character of the name means that the extended attribute is a standard system extended attribute (SEA), which is reserved for system use.

Various objects in the various file systems may or may not have extended attributes. The QSYS.LIB and independent ASP QSYS.LIB file systems support three predefined extended attributes: .SUBJECT, .TYPE, and .CODEPAGE. In the document library services (QDLS) file system, folders and documents can have any kind of extended attribute. Some folders and documents may have extended attributes, and some may not. In the “root” (/), QOpenSys, and user-defined file systems, all directories, stream files, and symbolic links can have extended attributes of any kind. Some, however, may not have any extended attributes at all.

- | The Work with Object Links (WRKLNK) command and Display Object Links (DSPLNK) command can be
- | used to display the .SUBJECT extended attribute for an object. There is no other integrated file system
- | support through which applications or users can access and change extended attributes. The only
- | exceptions to this rule are the Display a UDFS (DSPUDFS) and the Display Mounted File System
- | Information (DSPMFSINF) CL commands, which present extended attributes to users.

Extended attributes associated with some objects in the QDLS can, however, be changed through interfaces provided by the hierarchical file system (HFS).

If a client PC is connected to a System i platform through OS/2® or Windows, the programming interfaces of the respective operating system (such as DosQueryFileInfo and DosSetFileInfo) can be used to query and set the extended attributes of any file object. OS/2 users can also change the extended attributes of an object on the desktop by using the settings notebook; that is, by selecting Settings on the menu associated with the object.

If you define extended attributes, use the following naming guidelines:

- The name of an extended attribute can be up to 255 characters long.
- Do not use a period (.) as the first character of the name. An extended attribute whose name begins with a period is interpreted as a standard system extended attribute.

- To minimize the possibility of name conflicts, use a consistent naming structure for extended attributes. The following form is recommended:

`CompanyNameProductName.Attribute_Name`

Scanning support

With the i5/OS operating system, you can scan integrated file system objects.

This support creates flexibility for users by allowing scans for various items; users decide when the scan should occur and what actions to take based on the results of their scans.

The two exit-points related to this support are:

- **QIBM_QP0L_SCAN_OPEN** - Integrated File System Scan on Open Exit Program
For this exit point, the integrated file system scan on open exit program is called to do scan processing when an integrated file system object is opened under certain conditions.
- **QIBM_QP0L_SCAN_CLOSE** - Integrated File System Scan on Close Exit Program
For this exit point, the integrated file system scan on close exit program is called to do scan processing when an integrated file system object is closed under certain conditions.

Note: Only objects in file systems that have been fully converted to *TYPE2 directories will be scanned.

Related tasks

“Setting whether objects should be scanned or not” on page 125
Follow these steps to set whether an object should be scanned or not.

Related information

Integrated File System Scan on Open Exit Program

Integrated File System Scan on Close Exit Program

Examples: Scanning for viruses and files being opened

These examples show what the exit program can scan for.

- **Viruses**
The exit programs can scan for viruses. If a virus is located in a file, the antivirus program can act accordingly by repairing the problem or by attempting to quarantine the virus. Because the System i platform itself cannot be infected by the virus, what this accomplishes is a reduction in virus transmissions between systems.
- **Calls to know when a file was opened**
You may also scan to find out when a file was opened. By enacting this scan, you are able to track the date and time of when certain files were accessed. This is useful when you want to track the behavior of certain users.

The scan can occur at two different times depending on how the system values are set and how the scan environment is established. The following list describes different kinds of scanning depending on the time they occur.

1. Runtime scanning

A runtime scan is a scan on a file or files during normal day-to-day activities. This ensures the integrity of your files every time they are accessed. Scanning during your normal activities allows you to ensure that your file or files are current for whatever standards you are scanning.

Example of a runtime scan for viruses

You choose to access a file on the integrated file system from your PC. When the file is opened from the PC, it is scanned. Because an open exit program is registered and the QSCANFS system value is set to scan files in the "root" (/), QOpenSys and UDFS file systems. The scan shows one virus is

found and the anti-virus exit program proceeds to repair the problem. After the exit program repairs the file, the file is no longer infected. Thus, the access from the PC is not infected and it cannot spread the infection.

Now say that instead of scanning for viruses on that access, you choose not to do a runtime scan. Then, after accessing the infected file from your PC, the virus might be transferred to your PC. By employing a runtime scan, the virus can be detected before it spreads to your PC.

The main shortcoming of this method is that resource time is required to do the scans. Users attempting to access a file must wait until the scan is completed, before being able to use the file. The system ensures that scanning is performed only when required, not on every access.

2. Mass or manually activated scanning

You can use this option if you want to scan multiple items at the same time. In this instance, you can set the scan to occur when your system is offline, such as during the weekend. This has very little impact on accessing files during your normal day-to-day activities. The scan is done offline. Therefore, it can reduce runtime scan overhead for files that do not change after the mass scan is completed because re-scans are not required when such files are accessed again.

Related concepts

“Related system values”

Related to this scan support are two system values. You can use these two system values to establish the scanning environment you want for your system.

Related information

Integrated File System Scan on Open Exit Program

Integrated File System Scan on Close Exit Program

Related system values

Related to this scan support are two system values. You can use these two system values to establish the scanning environment you want for your system.

Listed as follows are the names of the two system values and the descriptions for each one. These system values and their control options are described for the iSeries Navigator. The comparable character-based interface values are listed in parenthesis following the iSeries Navigator names. For example, for the system value QSCANFSCTL, when the iSeries Navigator control option ‘Scan accesses through file servers only’ is selected, you would essentially be creating the same results by specifying the character-based control option *FSVROONLY.

The name and descriptions of these system values are as follows:

1. Use registered exit program to scan the "root" (/), QOpenSys, and user-defined file systems (QSCANFNS)

This system value can be used to specify whether file systems should be scanned. Only objects in the "root" (/), QOpenSys, and user-defined file systems will be scanned if the file system has been fully converted. This value specifies whether objects should be scanned by exit programs registered with any of the integrated file system scan-related exit points.

The default value is that objects will be scanned if any exit programs are registered.

2. Scan control (QSCANFSCTL)

For this system value, you can use the default control options or you can use specified control options. For brief descriptions on the different specified control options based on the iSeries Navigator system values, see the following list.

- Scan accesses through file servers only (*FSVROONLY specified)
A scan will only take place if you access the System i platform from a file server. If this option is not selected, all accesses will be scanned.
- Fail request if exit program fails (*ERRFAIL specified)

If there are errors when the exit program is called, the request or operation that triggered the call to the exit program will fail. If this option is not selected, the system will skip the failing exit program and the object will be treated as if it was not scanned.

- Perform write access upgrades (*NOWRTUPG not specified)

The access upgrade will occur for the scan descriptor passed to the exit program to include write access. If *NOWRTUPG option is not selected, the system will not attempt to do the write access upgrade.

If *NOWRTUPG is specified, the system will **not** attempt to upgrade the access for the scan descriptor passed to the exit program to include write access. If *NOWRTUPG is not specified, the system will attempt to do the write access upgrade.

- Use 'only when objects have changed' attribute to control scan (*USEOCOATR specified)

The 'object change only' attribute (only scan the object if it has been modified) will be used. If this option is not selected, this attribute will not be used and the object will be scanned after it is modified and when scan software indicates an update.

- Fail close request if scan fails during close (*NOFAILCLO not specified)

If an object failed a scan during close processing, the close request will be failed. If this option is not selected, the close request will not be failed. When not selected, this value overrides the specification of the 'fail request if exit program fails' value.

If *NOFAILCLO is specified, the system will **not** fail the close request with an indication of scan failure, even if the object failed a scan which was done as part of the close processing.

- Scan on next access after object has been restored (*NOPOSTRST not specified)

Objects will be scanned after they are restored. If 'the object will not be scanned' attribute is specified, the object will be scanned once after being restored. If the attribute 'object change only' is specified, the object will be scanned after being restored.

If *NOPOSTRST is specified while the objects are restored, they will not be scanned just because they were restored. If the object attribute is 'the object will not be scanned', the object will not be scanned at any time. If the object attribute is 'object change only', the object will only be scanned if it is modified after being restored.

Related information

Security system values: Use registered exit programs to scan the root (/), QOpenSys, and user-defined file systems

Security system values: Scan control

Scanning occurrences

Scanning can occur for a variety of reasons. Here is some information about when and why a scan might occur.

To see the current scan status and attribute of an object, you can use the Work with Object Links (WRKLNK) command, the Display Object Links (DSPLNK) command, the Get Attributes (Qp0lGetAttr()) API, or the Properties page in iSeries Navigator.

Related information

Work with Object Links (WRKLNK) command

Display Object Links (DSPLNK) command

Qp0lGetAttr()--Get Attributes API

Object change:

A scan would occur if the object is accessed after an object has changed or has been modified.

Normally, the modification occurs in the object's data. Examples of modifications to an object are writing to the object directly, or through memory mapping, truncating the object, or clearing the object. If the object's CCSID attribute changes, this will also trigger a scan on the next access.

Signature change:

A scan occurs when the object is accessed if the global signature is different from the object's signature.

The global or independent ASP group signatures represent the level of software associated with the scan-related exit programs. The object signature reflects the global or independent ASP signature when the object was last scanned. When an object is not in an independent ASP group, the object signature is compared to the global scan signature. If the object is in an independent ASP, the object signature is compared to the associated independent ASP group scan signature.

Note: In the following example, the phrases scan key and scan key signature are used. The scan key is a method to identify one set of scanning software. An example of this is for a specific company. The scan key signature allows the set of scanning software to indicate the level of support it provides. One example of this is a set of virus definitions.

Here is an example of when an object is not in an independent ASP group and a scan occurs:

1. An exit program is registered to the QIBM_QP0L_SCAN_OPEN exit point. A scan key and a scan key signature were specified as follows:

Scan key: XXXXXX
Scan key signature: 0000000000

The global scan signature is 0000 and is not updated.

2. An exit program is then registered to the QIBM_QP0L_SCAN_CLOSE exit point. A scan key and a scan key signature were specified as follows:

Scan key: XXXXXX
Scan key signature: 1111111111

The global scan signature is then updated to 0001.

3. Next, a file is opened that currently has an object signature of 0000. The existence of the exit programs, coupled with the difference in global scan signatures (0000 to 0001), initiates a scan. When the scan completes successfully, the file signature is updated to 0001.
4. If the file is opened by another user, it will not be re-scanned since the object and global signatures match.

The example below displays that the exit program wants to cause a re-scan to occur:

1. Support has been added to the system to scan for new types of viruses. The Change Scan Signature (QP0LCHSG) API is called to update the scan keys' scan key signature. A scan key and a scan key signature are specified as follows:

Scan key: XXXXXX
Scan key signature: 2222222222

The global scan key signature is then updated to 0002.

2. If the previously scanned file is now opened, the difference in signatures will cause a re-scan.

The example continues on to show when an object is in an independent ASP group:

1. An independent ASP is varied on for the first time and a file in the independent ASP is opened. When the first file is opened, the independent ASP scan key list is compared to the system scan key list. The two are different because of the fact that there is no independent ASP scan key list. In this case, the independent ASP scan key list obtains the global scan key list. The independent ASP scan key list then has a scan key of XXXXXX and a scan key signature of 2222222222. As a result, the independent ASP scan signature is changed to 0001. When the file in the independent ASP is opened

that currently has an object signature of 0000, it is then compared to the independent ASP scan signature of 0001, and because of the difference the file is scanned. When scanned successfully, the file signature is updated to 0001.

Note: A signature change will trigger a scan unless the object has the 'object change only' attribute and the *USEOCOATR system value specified.

Related information

- Integrated File System Scan on Open Exit Program
- Integrated File System Scan on Close Exit Program
- Change Scan Signature (QP0LCHSG) API

Different CCSID:

If an object is accessed with a different coded character set identifier (CCSID) than was previously scanned for that object, a scan would be triggered.

An example of this scan is when a file with data stored in CCSID 819 is opened in CCSID 1200, and scanned successfully. As long as the file's data is not changed, then every time that file is opened in CCSID 1200, a scan is not triggered. However, if that file is opened in a different CCSID, for example, 37, a scan is triggered for that CCSID 37. If that scan is also successful, then any subsequent accesses with CCSID 1200 and 37 will not trigger an additional scan.

Only two CCSIDs and one binary indication are kept in an effort to minimize data stored on the system. If you typically access the same object with many different CCSIDs, then this can cause a lot of additional scanning to occur.

During save operation:

This provides yet another example of when a scan might occur. A scan can be requested when an object is saved.

The Save Object (SAV) command now includes a SCAN parameter that allows for specification as to whether the files will be scanned when being saved. You may also request for the object not to be saved if it has either previously failed a scan or if it fails a scan during the save. This prevents files who fail the scan from being put on media and possibly moved to other systems.

Note: This does not mean that when your object is restored it will be marked as having been scanned. Whenever objects are restored, the entire scan status history is cleared.

Related information

- Save Object (SAV) command

Object integrity check:

Lastly, a scan can be requested if the SCANFS parameter on the Check Object Integrity (CHKOBJITG) command is specified with a value of *YES.

This option is ideal if you want to determine whether a file is good without opening it. If SCANFS (*STATUS) is specified, then all objects which have failed previous scans will log a scan failure violation.

Related information

- Change Object Integrity (CHGOBJITG) command

File systems

A *file system* provides you with the support to access specific segments of storage that are organized as logical units. These logical units on your system are files, directories, libraries, and objects.

Each file system has a set of logical structures and rules for interacting with information in storage. These structures and rules may be different from one file system to another. In fact, from the perspective of structures and rules, the i5/OS support for accessing database files and various other object types through libraries can be thought of as a file system. Similarly, the i5/OS support for accessing documents (which are really stream files) through the folders structure might be thought of as a separate file system.

The integrated file system treats the library support and folders support as separate file systems. Other types of file management support that have differing capabilities are also treated as separate file systems.

You can interact with any of the file systems through a common interface. This interface is optimized for the input and output of stream data, in contrast to the record input and output that is provided through the data management interfaces. The provided commands, menus and displays, and application programming interfaces (APIs) allow interaction with the file systems through this common interface.

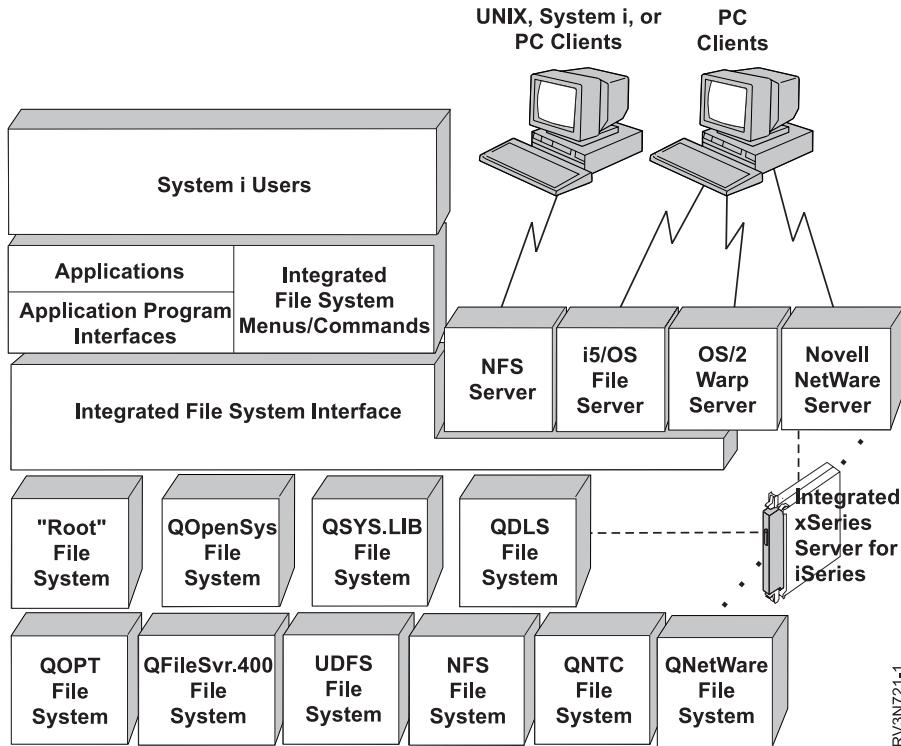


Figure 9. File systems, file servers, and the integrated file system interface

Using Network File Systems through the integrated file system interface

The Network File System (NFS) is accessible through the integrated file system interface. Be aware of these considerations and limitations.

Related information

Optical storage



Network file system support PDF

Planning integrated file system security

File system comparison

These tables summarize the features and limitations of each file system.

Table 2. File system summary (Part 1 of 2)

Capability	"root" (/)	QOpenSys	QSYS.LIB ¹⁶	QDLS	QNTC
Standard part of i5/OS	Yes	Yes	Yes	Yes	Yes
Type of file	Stream	Stream	Record ¹²	Stream	Stream
File size limit	T2=1 TB; T1=128 GB	T2=1 TB; T1=128 GB	Database file sizes	4 GB	Varies ¹⁷
Integrated with OfficeVision® (for example, file can be mailed)	No	No	No	Yes	No
Access through i5/OS file server	Yes	Yes	Yes	Yes	Yes
Direct access through file server I/O processor ¹	No	No	No	No	Yes
Comparative speed for open/close	Medium ²	Medium ²	Low ²	Low ²	Medium ²
Case-sensitive name search	No	Yes	No ⁴	No ⁵	No
Maximum length of each component in path name	255 char ¹⁹	255 char ¹⁹	10.6 char ⁶	8.3 char ⁷	255 char ¹⁹
Maximum length of path name ⁸	16MB	16MB	55 – 66 char ⁴	82 char	255 char
Maximum length of extended attributes for an object	2GB	2GB	Varies ⁹	32KB	0 ¹⁸
Maximum levels of directory hierarchy within file system	No limit ¹⁰	No limit ¹⁰	3	32	127
Maximum links per object ¹¹	Varies ¹⁵	Varies ¹⁵	1	1	1
Supports symbolic links	Yes	Yes	No	No	No
Object/file can have owner	Yes	Yes	Yes	Yes	No
Supports integrated file system commands	Yes	Yes	Yes	Yes	Yes
Supports integrated file system APIs	Yes	Yes	Yes	Yes	Yes
Supports hierarchical file system (HFS) APIs	No	No	No	Yes	No
Threadsafe ¹³	Yes	Yes	Yes	No	Yes
Supports object journaling	Yes	Yes	Yes ¹⁴	No	No

Table 2. File system summary (Part 1 of 2) (continued)

Capability	"root" (/)	QOpenSys	QSYS.LIB ¹⁶	QDLS	QNTC
Notes:					
<ol style="list-style-type: none"> The file server I/O processor is hardware used by LAN Server. When accessed through the i5/OS file server. When accessed through a LAN Server client PC. Access using i5/OS APIs is comparatively slow. The QSYS.LIB file system has a maximum path name length of 55 characters. The independent ASP QSYS.LIB file system has a maximum path length of 66 characters. See "Document library services file system (QDLS)" on page 45 for details. Up to 10 characters for the object name and up to 6 characters for the object type. Up to 8 characters for the name and 1 to 3 characters for the file type extension (if any). Assuming an absolute path name that begins with / followed by the file system name (such as /QDLS...). The QSYS.LIB and independent ASP QSYS.LIB file systems support three predefined extended attributes: .SUBJECT, .CODEPAGE, and .TYPE. The maximum length is determined by the combined length of these three extended attributes. In practice, directory levels are limited by program and system space limits. Except a directory, which can have only one link to another directory. The user spaces in QSYS.LIB and independent ASP QSYS.LIB file systems support stream file input and output. Integrated file system APIs are threadsafe when the operation is directed to an object that resides in a threadsafe file system. When these APIs are operating on objects in file systems that are not threadsafe when multiple threads are running in the job, the API will fail. QSYS.LIB and independent ASP QSYS.LIB file systems support journaling different object types than the "root" (/), UDFS, and QOpenSys file systems. *TYPE2 directories have a limit of one million links per object and a limit of 999 998 subdirectories. *TYPE1 directories have a limit of 32 767 links per object. Data in this column refers to both the QSYS.LIB file system and the independent ASP QSYS.LIB file system. Depends on the system being accessed. QNTC does not support extended attributes. For certain CCSID values, the maximum length can be less than 255 characters. 					
Abbreviations					
<ul style="list-style-type: none"> char = characters T1 = *TYPE1 *STMF T2 = *TYPE2 *STMF B = bytes KB = kilobytes MB = megabytes GB = gigabytes TB = terabytes 					

Table 3. File system summary (Part 2 of 2)

Capability	QOPT	QFileSvr.400	UDFS	NFS	QNetWare
Standard part of i5/OS	Yes	Yes	Yes	Yes	No
Type of file	Stream	Stream	Stream	Stream	Stream
File size limit	4 GB	2 GB - 1	T2 = 1 TB; T1=128 GB	Varies ¹⁶	2 GB
Integrated with OfficeVision (for example, file can be mailed)	No	No	No	No	No
Access through i5/OS file server	Yes	Yes	Yes	Yes	Yes
Direct access through the Integrated PC Server ¹	No	No	No	No	Yes

Table 3. File system summary (Part 2 of 2) (continued)

Capability	QOPT	QFileSvr.400	UDFS	NFS	QNetWare
Comparative speed for open/close	Low	Low ²	Medium ²	Medium ²	High ¹¹
Case-sensitive name search	No	No ²	Yes ¹²	Varies ²	No
Maximum length of each component in path name	Varies ⁴	Varies ²	255 char ¹⁷	Varies ²	255 char ^{13, 17}
Maximum length of path name	294 char	No limit ²	16MB	No limit ²	255 char
Maximum length of extended attributes for an object	8MB	0 ⁶	2GB ¹⁰	0 ⁶	64KB
Maximum levels of directory hierarchy within file system	No limit ⁷	No limit ²	No limit ⁷	No limit ²	100
Maximum links per object ⁷	1	1	Varies ¹⁵	Varies ²	1
Supports symbolic links	No	No	Yes	Yes ²	No
Object/file can have owner	No	No ⁹	Yes	Yes ²	Yes
Supports integrated file system commands	Yes	Yes	Yes	Yes	Yes
Supports integrated file system APIs	Yes	Yes	Yes	Yes	Yes
Supports hierarchical file system (HFS) APIs	Yes	No	No	No ²	No
Threadsafe ¹⁴	Yes	Yes	Yes	Yes	No
Supports object journaling	No	No	Yes	No	No

Table 3. File system summary (Part 2 of 2) (continued)

Capability	QOPT	QFileSvr.400	UDFS	NFS	QNetWare
Notes:					
<ol style="list-style-type: none"> 1. The file server I/O processor is hardware used by LAN Server. 2. Depends on which remote file system is being accessed. 3. When accessed through the i5/OS file server. 4. See "Optical file system (QOPT)" on page 48 for details. 5. Assuming an absolute path name that begins with / followed by the file system name. 6. The QFileSvr.400 file system does not return extended attributes even if the file system being accessed supports extended attributes. 7. In practice, directory levels are limited by program and system space limits. 8. Except a directory, which can have only one link to another directory. 9. The file system being accessed may support object owners. 10. The maximum length of extended attributes for the UDFS itself cannot exceed 40 bytes. 11. When accessed through a Novell NetWare client PC. Access using i5/OS APIs is comparatively slow. 12. Case-sensitivity can be specified when a UDFS is created. If the *MIXED parameter is used when creating a UDFS, it will allow a case-sensitive search. 13. NetWare Directory Services objects are a maximum of 255 characters. Files and directories are limited to DOS 8.3 format. 14. Integrated file system APIs are threadsafe when they are accessed in a multi-thread capable process. The file system does not allow accesses to the file systems that are not threadsafe. 15. *TYPE2 directories have a limit of one million links per object. *TYPE1 directories have a limit of 32 767 links per object. 16. Depends on the system being accessed. 17. For certain CCSID values, the maximum length can be less than 255 characters. 					
Abbreviations					
<ul style="list-style-type: none"> • char = characters • T1 = *TYPE1 *STMF • T2 = *TYPE2 *STMF • B = bytes KB = kilobytes MB = megabytes GB = gigabytes TB = terabytes 					

Related reference

"root" (/) file system" on page 29

The "root" (/) file system takes full advantage of the stream file support and hierarchical directory structure of the integrated file system. It has the characteristics of the DOS and OS/2 file systems.

"Open systems file system (QOpenSys)" on page 31

The QOpenSys file system is compatible with open system standards based on UNIX, such as POSIX and X/Open Portability Guide (XPG). Like the "root" (/) file system, this file system takes advantage of the stream file and directory support that is provided by the integrated file system.

"User-defined file systems (UDFSs)" on page 33

The user-defined file systems (UDFSs) reside on the auxiliary storage pool (ASP) or independent auxiliary storage pool (ASP) of your choice. You can create and manage these file systems.

"Library file system (QSYS.LIB)" on page 39

The QSYS.LIB file system supports the i5/OS library structure.

"Independent ASP QSYS.LIB" on page 42

The independent ASP QSYS.LIB file system supports the i5/OS library structure in independent auxiliary storage pools (ASPs) you create and define. This file system provides access to database files and all of the other i5/OS object types that the library support manages in the independent ASPs.

"Document library services file system (QDLS)" on page 45

The QDLS file system supports the folders structure. It provides access to documents and folders.

“Optical file system (QOPT)” on page 48

The QOPT file system provides access to stream data that is stored on optical media.

“NetWare file system (QNetWare)” on page 50

The QNetWare file system provides access to data on stand-alone PC servers that are running Novell NetWare 5.1 or 6.0.

“iSeries NetClient file system (QNTC)” on page 53

The QNTC file system provides access to data and objects that are stored on an Integrated xSeries Server for iSeries running Windows NT 4.0 Server or later, or Linux®. The QNTC file system also provides access to data and objects that are stored on remote servers running Windows NT 4.0 or later, Linux Samba 3.0 or later, or supported versions of iSeries NetServer™.

“i5/OS file server file system (QFileSvr.400)” on page 57

The QFileSvr.400 file system provides transparent access to other file systems that reside on remote System i platforms. It is accessed through a hierarchical directory structure.

“Network File System (NFS)” on page 61

The Network File System (NFS) provides the user with access to data and objects that are stored on a remote NFS server.

Related information

Journal management

“root” (/) file system

The “root” (/) file system takes full advantage of the stream file support and hierarchical directory structure of the integrated file system. It has the characteristics of the DOS and OS/2 file systems.

In addition, it:

- Is optimized for stream file input and output.
- Supports multiple hard links and symbolic links.
- Supports local sockets.
- Supports threadsafe APIs.
- Supports *FIFO objects.
- Supports the /dev/null and /dev/zero *CHRSF objects as well as other *CHRSF objects.
- Supports the journaling of object changes.
- Supports the scanning of objects using the integrated file system scan-related exit points.

The “root” (/) file system has support for the character special files (*CHRSF) called /dev/null and /dev/zero. Character special files are associated with a device or resource of a computer system. They have path names that appear in directories and have the same access protection as regular files. The /dev/null or /dev/zero character special files are always empty, and any data written to /dev/null or /dev/zero is discarded. The files /dev/null and /dev/zero have an object type of *CHRSF and can be used like regular files, except that no data is ever read in the /dev/null file, and the /dev/zero file always returns successfully with the data cleared to zeros.

Accessing the “root” (/) file system

The “root” (/) file system can be accessed through the integrated file system interface using either the i5/OS file server or the integrated file system commands, user displays, and APIs.

Case-sensitivity in the “root” (/) file system

The file system preserves the same uppercase and lowercase form in which object names are entered, but no distinction is made between uppercase and lowercase when the system searches for names.

Path names in the "root" (/) file system

Path names have specific form in the "root" (/) file system.

| /Directory/Directory . . . /Object

- Each component of the path name can be up to 255 characters long, much longer than in the QSYS.LIB or QDLS file systems. The full path name can be extremely long, up to 16 megabytes.
- There is no limit to the depth of the directory hierarchy other than program and system space limits.
- The characters in names are converted to UCS2 Level 1 form (for *TYPE1 directories) and UTF-16 (for *TYPE2 directories) when the names are stored.

Related concepts

"Name continuity" on page 17

When you use the "root" (/), QOpenSys, and user-defined file systems, you can take advantage of system support that ensures characters in object names remain the same.

"*TYPE2 directories" on page 10

The "root" (/), QOpenSys, and user-defined file systems (UDFS) in the integrated file system support the *TYPE2 directory format. The *TYPE2 directory format is an enhancement of the original *TYPE1 directory format.

"Path name" on page 14

A *path name* (also called a *pathname* on some systems) tells the system how to locate an object.

Links in the "root" (/) file system

Multiple hard links to the same object are allowed in the "root" (/) file system. Symbolic links are fully supported.

A symbolic link can be used to link from the "root" (/) file system to an object in another file system, such as QSYS.LIB, Independent ASP QSYS.LIB, or QDLS.

Related concepts

"Link" on page 11

A *link* is a named connection between a directory and an object. A user or a program can tell the system where to find an object by specifying the name of a link to the object. A link can be used as a path name or as part of a path name.

Integrated file system commands in the "root" (/) file system

All of the commands listed in the Accessing using CL commands topic and the displays described in the Accessing using menus and displays topic can operate on the "root" (/) file system. However, it might not be safe to use these commands in a multithread-capable process.

Related tasks

"Accessing using menus and displays" on page 64

You can perform operations on files and other objects in the integrated file system by using a set of menus and displays provided by your system.

Related reference

"Accessing using CL commands" on page 65

All of the operations that you can do through the integrated file system menus and displays can be done by entering control language (CL) commands. These commands can operate on files and other objects in any file system that are accessible through the integrated file system interface.

Integrated file system APIs in the "root" (/) file system

All of the APIs listed in the Performing operations using APIs topic can operate on the "root" (/) file system.

Related reference

"Performing operations using APIs" on page 104

Many of the application programming interfaces (APIs) that perform operations on integrated file system objects are in the form of C language functions.

Related information

Application programming interfaces (APIs)

Object changes journaling in the "root" (/) file system

Objects in the "root" (/) file system can be journaled. This function enables you to recover the changes to an object that have occurred since the object was last saved.

Related concepts

"Journaling objects" on page 90

The primary purpose of journaling is to enable you to recover the changes to an object that have occurred since the object was last saved. Additionally, a key use of journaling is to assist in the replication of object changes to another system either for high availability or workload balancing.

UDP and TCP devices in the "root" (/) file system

The "root" (/) file system under the directory of /dev/xti will now hold two device drivers named udp and tcp.

Both of the drivers are character special files (*CHRFSs) and are created during the first initial program load (IPL). The User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) device drivers are used to open a connection to the UDP and TCP transport providers. Both of these drivers are user devices and receive a new device major number. They also have cloned open operations, which means that each open operation obtains a unique instance of the device. The use of these devices is only supported in the i5/OS Portable Application Solutions Environment (PASE). The following table contains the objects that will be created and their properties.

Table 4. Device driver objects and properties

Path name	Type	Major	Minor	Owner	Owner data authorities	Group	Group data authorities	Public data authorities
/dev/xti	*DIR	N/A	N/A	QSYS	*RWX	None	*RX	*RX
/dev/xti/tcp	*CHRFS	Clone	TCP	QSYS	*RW	None	*RW	*RW
/dev/xti/udp	*CHRFS	Clone	UDP	QSYS	*RW	None	*RW	*RW

Related information

i5/OS PASE

Open systems file system (QOpenSys)

The QOpenSys file system is compatible with open system standards based on UNIX, such as POSIX and X/Open Portability Guide (XPG). Like the "root" (/) file system, this file system takes advantage of the stream file and directory support that is provided by the integrated file system.

In addition, it:

- Is accessed through a hierarchical directory structure similar to UNIX systems.
- Is optimized for stream file input and output.
- Supports multiple hard links and symbolic links.
- Supports case-sensitive names.
- Supports local sockets.
- Supports threadsafe APIs.
- Supports *FIFO objects.
- Supports the journaling of object changes.
- Supports the scanning of objects using the integrated file system scan-related exit points.

The QOpenSys file system has the same characteristics as the “root” (/) file system, except it is case-sensitive to enable support for UNIX-based open systems standards.

Accessing QOpenSys

QOpenSys can be accessed through the integrated file system interface using either the i5/OS file server or the integrated file system commands, user displays, and APIs.

Case-sensitivity in the QOpenSys file system

Unlike the “root” (/) file system, the QOpenSys file system distinguishes between uppercase or lowercase characters when searching for object names.

For example, a character string supplied in all uppercase characters will not match the same character string in which any of the characters is lowercase.

This case-sensitivity allows you to use duplicate names, provided there is some difference in uppercase and lowercase of the characters making up the name. For example, you can have an object named Payro11, an object named PayRo11, and an object named PAYROLL in the same directory in QOpenSys.

Path names in the QOpenSys file system

Path names have specific form in the QOpenSys file system.

```
| /QOpenSys/Directory/Directory/ . . . /Object
```

- Each component of the path name can be up to 255 characters long. The full path name can be up to 16 MB long.
- There is no limit to the depth of the directory hierarchy other than program and system space limits.
- The characters in names are converted to UCS2 Level 1 form (for *TYPE1 directories) and UTF-16 (for *TYPE2 directories) when the names are stored.

Related concepts

“Name continuity” on page 17

When you use the “root” (/), QOpenSys, and user-defined file systems, you can take advantage of system support that ensures characters in object names remain the same.

“*TYPE2 directories” on page 10

The “root” (/), QOpenSys, and user-defined file systems (UDFS) in the integrated file system support the *TYPE2 directory format. The *TYPE2 directory format is an enhancement of the original *TYPE1 directory format.

“Path name” on page 14

A *path name* (also called a *pathname* on some systems) tells the system how to locate an object.

Links in the QOpenSys file system

Multiple hard links to the same object are allowed in the QOpenSys file system. Symbolic links are fully supported.

A symbolic link can be used to link from the QOpenSys file system to an object in another file system.

Related concepts

“Link” on page 11

A *link* is a named connection between a directory and an object. A user or a program can tell the system where to find an object by specifying the name of a link to the object. A link can be used as a path name or as part of a path name.

Integrated file system commands and displays in the QOpenSys file system

All of the commands that are listed in the Accessing using CL commands topic and the displays that are described in the Accessing using menus and displays topic can operate on the QOpenSys file system. However, it may not be safe to use these commands in a multi-thread capable process.

Related tasks

“Accessing using menus and displays” on page 64

You can perform operations on files and other objects in the integrated file system by using a set of menus and displays provided by your system.

Related reference

“Accessing using CL commands” on page 65

All of the operations that you can do through the integrated file system menus and displays can be done by entering control language (CL) commands. These commands can operate on files and other objects in any file system that are accessible through the integrated file system interface.

Integrated file system APIs in the QOpenSys file system

All the APIs listed in the Performing operations using APIs topic can operate on the QOpenSys file system.

Related reference

“Performing operations using APIs” on page 104

Many of the application programming interfaces (APIs) that perform operations on integrated file system objects are in the form of C language functions.

Related information

Application programming interfaces (APIs)

Object changes journaling in the QOpenSys file system

Objects in the QOpenSys file system can be journaled. This function enables you to recover the changes to an object that have occurred since the object was last saved.

Related concepts

“Journaling objects” on page 90

The primary purpose of journaling is to enable you to recover the changes to an object that have occurred since the object was last saved. Additionally, a key use of journaling is to assist in the replication of object changes to another system either for high availability or workload balancing.

User-defined file systems (UDFSs)

The user-defined file systems (UDFSs) reside on the auxiliary storage pool (ASP) or independent auxiliary storage pool (ASP) of your choice. You can create and manage these file systems.

In addition, they:

- Provide a hierarchical directory structure similar to PC operating systems such as DOS and OS/2
- Are optimized for stream file input and output
- Support multiple hard links and symbolic links
- Support local sockets
- Support threadsafe APIs
- Support *FIFO objects
- Support the journaling of object changes
- Support the scanning of objects using the integrated file system scan-related exit points

You can create multiple UDFSs by giving each a unique name. You can specify other attributes for a UDFS during its creation, including:

- An ASP number or independent ASP name where the objects that are located in the UDFS are stored.
- The case-sensitivity characteristics of the object names that are located within a UDFS.

The case-sensitivity of a UDFS determines whether uppercase and lowercase characters will match when searching for object names within the UDFS.

- The create object scanning attribute which defines what the scan attribute should be for objects created in a UDFS.

- The restrict rename and unlink attribute.
- The auditing value for a UDFS.
- The different stream file formats, *TYPE1 and *TYPE2.

User-defined file system concepts

In a user-defined file system (UDFS), as in the “root” (/) and QOpenSys file systems, you can create directories, stream files, symbolic links, local sockets, and *FIFO objects.

A single block special file object (*BLKSF) represents a UDFS. As you create UDFSs, you also automatically create block special files. The block special file is only accessible to the user through the integrated file system generic commands, APIs, and the QFileSvr.400 interface.

A UDFS exists only in two states: **mounted** and **unmounted**. When you mount a UDFS, the objects within it are accessible. When you unmount a UDFS, the objects within it become inaccessible.

In order to access the objects within a UDFS, you must mount the UDFS on a directory (for example, /home/JON). When you mount a UDFS on a directory, the original contents of that directory, including objects and subdirectories, become inaccessible. When you mount a UDFS, the contents of the UDFS become accessible through the directory path that you mount the UDFS over. For example, the /home/JON directory contains a file /home/JON/payroll. A UDFS contains three directories mail, action, and outgoing. After mounting the UDFS on /home/JON, the /home/JON/payroll file is inaccessible, and the three UDFS directories become accessible as /home/JON/mail, /home/JON/action, and /home/JON/outgoing. After unmounting the UDFS, the /home/JON/payroll file is accessible again, and the three directories in the UDFS become inaccessible. An initial program load (IPL) of the system unmounts all UDFSs. Therefore, the UDFSs need to be remounted after any IPL.

Note: A UDFS on an independent ASP cannot be mounted over.

To learn more about mounting file systems, see OS/400 Network File System Support  .

Accessing a user-defined file system through the integrated file system interface

A user-defined file system (UDFS) can be accessed through the integrated file system interface using either the i5/OS file server or the integrated file system commands, user displays, and APIs.

In using the integrated file system interface, you should be aware of the following considerations and limitations.

Related concepts

“Link” on page 11

A *link* is a named connection between a directory and an object. A user or a program can tell the system where to find an object by specifying the name of a link to the object. A link can be used as a path name or as part of a path name.

“Stream file” on page 16

A *stream file* is a randomly accessible sequence of bytes, with no further structure imposed by the system.

Related information

Create User-Defined FS (CRTUDFS) command

Case-sensitivity in an integrated file system user-defined file system

You can specify whether object names in the user-defined file system (UDFS) are case-sensitive or case-insensitive when you create it.

When you select case-sensitivity, uppercase and lowercase characters are distinguished when searching for object names. For example, a name that is supplied in all uppercase characters will not match the same name in which any of the characters are lowercase. Therefore, /home/MURPH/ and /home/murph/ are recognized as different directories. To create a case-sensitive UDFS, you can specify *MIXED for the CASE parameter when using the Create User-Defined File System (CRTUDFS) command.

When you select case-insensitivity, the system does not distinguish between uppercase and lowercase characters during searches for names. Therefore, the system recognizes /home/CAYCE and /HOME/cayce as the same directory, not as two separate directories. To create a case-insensitive UDFS, you can specify *MONO for the CASE parameter when using the CRTUDFS command.

In either case, the file system saves the same uppercase and lowercase forms in which the user enters object names. The case-sensitivity option only applies to how the user searches for names through the system.

Related information

Create User-Defined FS (CRTUDFS) command

Path names in an integrated file system user-defined file system

A block special file (*BLKSF) represents a user-defined file system (UDFS) when the entire UDFS and all of the objects within it need to be manipulated.

If your UDFS resides on the system or on a basic user ASP, block special file names must be of the form /dev/QASPXX/udfs_name.udfs

where XX is the ASP number where you store the UDFS, and udfs_name is the unique name of the UDFS within that ASP. Note that the UDFS name must end with the .udfs extension.

If your UDFS resides on an independent ASP, block special file names must be of the form /dev/asp_name/udfs_name.udfs

where asp_name is the name of independent ASP where you store the UDFS and udfs_name is the unique name of the UDFS within that independent ASP. Note that the UDFS name must end with the .udfs extension.

Path names for objects within a UDFS are relative to the directory over which you mount a UDFS. For example, if you mount the UDFS /dev/qasp01/wysocki.udfs over /home/dennis, then the path names for all objects within the UDFS will begin with /home/dennis.

Additional path name rules:

- Each component of the path name can be up to 255 characters long. The full path name can be up to 16 MB long.
- There is no limit to the depth of the directory hierarchy other than program and server space limits.
- The characters in names are converted to UCS2 Level 1 form (for *TYPE1 directories) and UTF-16 (for *TYPE2 directories) when the names are stored.

Related concepts

“Name continuity” on page 17

When you use the “root” (/), QOpenSys, and user-defined file systems, you can take advantage of system support that ensures characters in object names remain the same.

“*TYPE2 directories” on page 10

The “root” (/), QOpenSys, and user-defined file systems (UDFS) in the integrated file system support the *TYPE2 directory format. The *TYPE2 directory format is an enhancement of the original *TYPE1 directory format.

“Path name” on page 14

A *path name* (also called a *pathname* on some systems) tells the system how to locate an object.

Links in an integrated file system user-defined file system

A user-defined file system (UDFS) allows multiple hard links to the same object and fully supports symbolic links.

A symbolic link can create a link from a UDFS to an object in another file system.

Related concepts

“Link” on page 11

A *link* is a named connection between a directory and an object. A user or a program can tell the system where to find an object by specifying the name of a link to the object. A link can be used as a path name or as part of a path name.

Integrated file system commands in a user-defined file system

All of the commands that are listed in the Accessing using CL commands topic and the displays that are described in the Accessing using menus and displays topic can operate on a user-defined file system.

There are some CL commands that are specific to the UDFS and other mounted file systems in general. The following table describes them.

Table 5. User-defined file system CL commands

Command	Description
ADDMFS	Add Mounted File System. Places exported, remote server file systems over local client directories.
CRTUDFS	Create UDFS. Creates a user-defined file system.
DLTUDFS	Delete UDFS. Deletes a user-defined file system.
DSPMFSINF	Display Mounted File System Information. Displays information about a mounted file system.
DSPUDFS	Display UDFS. Displays information about a user-defined file system.
MOUNT	Mount a File System. Places exported, remote server file systems over local client directories. This command is an alias for the ADDMFS command.
RMVMFS	Remove Mounted File System. Removes exported, remote server file systems from the local client namespace.
UNMOUNT	Unmount a File System. Removes exported, remote server file systems from the local client namespace. This command is an alias for the RMVMFS command.

Note: You must mount a UDFS before any integrated file system commands can operate on the objects that are stored in that UDFS.

Related tasks

“Accessing using menus and displays” on page 64

You can perform operations on files and other objects in the integrated file system by using a set of menus and displays provided by your system.

Related reference

“Accessing using CL commands” on page 65

All of the operations that you can do through the integrated file system menus and displays can be done by entering control language (CL) commands. These commands can operate on files and other objects in any file system that are accessible through the integrated file system interface.

Integrated file system APIs in a user-defined file system

All of the APIs that are listed in Perform operations using APIs topic can operate on a user-defined file system.

Note: You must mount a UDFS before any integrated file system APIs can operate on the objects that are stored in that UDFS.

Related reference

“Performing operations using APIs” on page 104

Many of the application programming interfaces (APIs) that perform operations on integrated file system objects are in the form of C language functions.

Related information

Application programming interfaces (APIs)

Graphical user interface for a user-defined file system

iSeries Navigator, a graphical user interface on your PC, provides easy and convenient access to user-defined file systems (UDFSs).

This interface enables you to create, delete, display, mount, and unmount a UDFS from a Windows client.

You can perform operations on a UDFS through iSeries Navigator. Basic tasks include:

- “Creating a new user-defined file system” on page 124
- “Mounting a user-defined file system” on page 124
- “Unmounting a user-defined file system” on page 125

Creating an integrated file system user-defined file system

The Create User-Defined File System (CRTUDFS) command creates a file system that can be made visible through the integrated file system namespace, APIs, and CL commands.

The ADDMFS or MOUNT commands place the user-defined file system (UDFS) on top of the already existing local directory. You can create a UDFS in an ASP or independent ASP of your choice.

You can also specify the following items for a UDFS:

- Case-sensitivity
- Whether objects created in the UDFS should be scanned or not
- The auditing value for objects created in the UDFS
- The value for the restricted, rename and unlink attribute

Related information

Create User-Defined FS (CRTUDFS) command

Add Mounted FS (ADDMFS) command

Deleting an integrated file system user-defined file system

The Delete User-Defined File System (DLTUDFS) command deletes an existing, unmounted user-defined file system (UDFS), and all the objects within it.

The command will fail if you have mounted the UDFS. Deletion of a UDFS will cause the deletion of all objects in the UDFS. If you do not have appropriate authority to delete all of the objects within a UDFS, then none of the objects will be deleted.

Related information

Delete User-Defined FS (DLTUDFS) command

Displaying an integrated file system user-defined file system

The Display User-Defined File System (DSPUDFS) command presents the attributes of an existing user-defined file system (UDFS), whether mounted or unmounted.

The Display Mounted File System Information (DSPMFSINF) command will also present information about a mounted UDFS as well as any mounted file system.

Related information

Display User-Defined FS (DSPUDFS) command

Display Mounted FS Information (DSPMFSINF) command

Mounting an integrated file system user-defined file system

The Add Mounted File System (ADDMFS) and MOUNT commands make the objects in a file system accessible to the integrated file system namespace.

To mount a user-defined file system (UDFS), you need to specify *UDFS for the TYPE parameter on the ADDMFS command.

Note: A UDFS on an independent ASP cannot be mounted over.

Related information

Add Mounted FS (ADDMFS) command

Unmounting an integrated file system user-defined file system

The unmount command makes the contents of a user-defined file system (UDFS) inaccessible to the integrated file system interfaces.

The objects in a UDFS will not be individually accessible once the UDFS is unmounted. The Remove Mounted File System (RMVMFS) or UNMOUNT commands will make a mounted file system inaccessible to the integrated file system namespace. If any of the objects in the file system is in use (for example, a file is opened) at the time of using the command, you will receive an error message. The UDFS will remain mounted. If you have mounted over any part of the UDFS, then this UDFS cannot be unmounted until it is uncovered.

For example, you mount a UDFS /dev/qasp02/jenn.udfs over /home/judy in the integrated file system namespace. If you then mount another file system /pubs over /home/judy, then the contents of jenn.udfs will become inaccessible. Furthermore, you cannot unmount jenn.udfs until you unmount the second file system from /home/judy.

Note: A UDFS on an independent ASP cannot be mounted over.

Related information

Remove Mounted FS (RMVMFS) command

Saving and restoring an integrated file system user-defined file system

You can save and restore all user-defined file system (UDFS) objects, as well as their associated authorities.

The Save Object (SAV) command allows you to save objects in a UDFS, whereas the Restore Object (RST) command allows you to restore UDFS objects. Both commands function whether the UDFS is mounted or unmounted. However, to correctly save the UDFS attributes, and not just the objects within the UDFS, the UDFS should be unmounted.

Related information

Save Object (SAV) command

Restore Object (RST) command

Object changes journaling in a user-defined file system

Objects in user-defined file systems (UDFSs) can be journaled. This function enables you to recover the changes to an object that have occurred since the object was last saved.

Related concepts

“Journaling objects” on page 90

The primary purpose of journaling is to enable you to recover the changes to an object that have occurred since the object was last saved. Additionally, a key use of journaling is to assist in the replication of object changes to another system either for high availability or workload balancing.

User-defined file system and independent auxiliary storage pools

When you vary on an independent auxiliary storage pool (ASP), several changes occur within the “root” (/) file system.

These changes are:

- A directory is created inside the /dev directory for the independent ASP. The name of this directory matches the name of the device description associated with the ASP. If this directory exists before the vary on request, and this directory is not empty, then the vary on will proceed but you will not be able to work with any UDFS’s on the ASP. If this occurs, vary off the independent ASP, either rename the directory or remove its contents and then try the vary on request again.
- Within the /dev/asp_name directory you will find the block special file objects associated with all UDFS’s that reside on the independent ASP. There will always be a system-provided default UDFS. The path to the default UDFS’s block special file is: /dev/asp_name/QDEFAULT.UDFS
- The default UDFS is mounted over the directory /asp_name. The /asp_name directory does not need to exist before the vary on request. However, if it does exist, it must be empty. If it is not empty, the ASP will still be varied on, but the default UDFS will not be mounted. If this occurs, either rename the directory or remove its contents and then either vary off and try the vary on again or use the MOUNT command to mount the default UDFS.
- If the independent ASP is either a primary or secondary ASP, and the default UDFS was successfully mounted, then an additional file system will be mounted. The independent ASP QSYS.LIB file system will be mounted over /asp_name/QSYS.LIB.

Note: This file system cannot be mounted or unmounted independently of the default UDFS. It will always be mounted or unmounted automatically.

Related reference

“Independent ASP QSYS.LIB” on page 42

The independent ASP QSYS.LIB file system supports the i5/OS library structure in independent auxiliary storage pools (ASPs) you create and define. This file system provides access to database files and all of the other i5/OS object types that the library support manages in the independent ASPs.

Library file system (QSYS.LIB)

The QSYS.LIB file system supports the i5/OS library structure.

This file system provides you with access to database files and all of the other i5/OS object types that the library support manages on the system and in the basic user auxiliary storage pools (ASPs).

In addition, it:

- Supports all user interfaces and programming interfaces that operate on i5/OS libraries and objects in those libraries
- Supports all programming languages and facilities that operate on database files
- Provides extensive administrative support for managing i5/OS objects
- Supports stream I/O operations on physical file members, user spaces, and save files

Before the integrated file system was introduced in Version 3 of OS/400, the QSYS.LIB file system was the only file system. Programmers who used languages, such as RPG or COBOL, and facilities, such as DDS, to develop applications were using the QSYS.LIB file system. System operators who used commands, menus, and displays to manipulate output queues were using the QSYS.LIB file system, as were system administrators who were creating and changing user profiles.

All of these facilities and the applications based on these facilities work as they did before the introduction of the integrated file system. These facilities cannot, however, access QSYS.LIB through the integrated file system interface.

Accessing QSYS.LIB through the integrated file system interface

The QSYS.LIB file system can be accessed through the integrated file system interface using either the i5/OS file server or the integrated file system commands, user displays, and APIs.

QPWFSEVER authorization list in the QSYS.LIB file system

The QPWFSEVER is an authorization list (object type *AUTL) that provides additional access requirements for all objects in the QSYS.LIB file system being accessed through remote clients.

The authorities specified in this authorization list apply to all objects within the QSYS.LIB file system.

The default authority to this object is PUBLIC *USE authority. The administrator can use the EDTAUTL (Edit Authorization List) or WRKAUTL (Work With Authorization List) commands to change the value of this authority. The administrator can assign PUBLIC *EXCLUDE authority to the authorization list so that the general public cannot access QSYS.LIB objects from remote clients.

File-handling restrictions in the QSYS.LIB file system

Here are some restrictions to be aware of when handling files in the QSYS.LIB file system.

- Logical files are not supported.
- Physical files supported for text mode access are program-described physical files containing a single field and source physical files containing a single text field. Physical files supported for binary mode access include externally-described physical files in addition to those files supported for text mode access.
- Byte-range locking is not supported. For more information about byte-range locking, see the `fcntl()`--Perform File Control Command topic .
- If any job has a database file member open, only one job is given write access to that file member at any time. Other requests are allowed only read access.

Support for user spaces in the QSYS.LIB file system

QSYS.LIB supports stream input and output operations to user space objects.

For example, a program can write stream data to a user space and read data from a user space. The maximum size of a user space is 16 776 704 bytes.

Be aware that user spaces are not tagged with a CCSID (coded character set identifier). Therefore, the CCSID returned is the default CCSID of the job.

Support for save files in the QSYS.LIB file system

The QSYS.LIB file system supports stream I/O operations to save file objects.

For example, an existing save file has data that may be read out or copied to another file until it is necessary to place the data into a different, existing, and empty save file object. When a save file is open for writing, no other open instances of the file are allowed. A save file **does** allow multiple open instances

for reading, provided no job has more than one open instance of the file for reading. A save file may not be opened for read/write access. Stream I/O operations to save file data are not allowed when multiple threads are running in a job.

Stream I/O operations on a save file are not supported when the save file or its directory are being exported through the Network File System. They can, however, be accessed from PC clients and through the QFileSvr.400 file system.

Case-sensitivity in the QSYS.LIB file system

In general, the QSYS.LIB file system does not distinguish between uppercase and lowercase characters in the names of objects.

A search for object names achieves the same result regardless of whether characters in the names are uppercase or lowercase.

However, if a name is enclosed in quotation marks, the case of each character in the name is preserved. A search involving quoted names, therefore, is sensitive to the case of the characters in the quoted name.

Path names in the QSYS.LIB file system

Each component of the path name must contain the object name followed by the object type of the object.

- For example:

```
/QSYS.LIB/QGPL.LIB/PRT1.OUTQ
```

```
/QSYS.LIB/EMP.LIB/PAY.FILE/TAX.MBR
```

The object name and object type are separated by a period (.). Objects in a library can have the same name if they are different object types, so the object type must be specified to uniquely identify the object.

- The object name in each component can be up to 10 characters long, and the object type can be up to 6 characters long.
- The directory hierarchy within QSYS.LIB can be either two or three levels deep (two or three components in the path name), depending on the type of object being accessed. If the object is a database file, the hierarchy can contain three levels (library, file, member); otherwise, there can be only two levels (library, object). The combination of the length of each component name and the number of directory levels determines the maximum length of the path name.
If "root" (/) and QSYS.LIB are included as the first two levels, the directory hierarchy for QSYS.LIB can be up to five levels deep.
- The characters in names are converted to CCSID 37 when the names are stored. Quoted names, however, are stored using the CCSID of the job.

For more information about CCSIDs, see the i5/OS globalization topic.

Related concepts

"Path name" on page 14

A *path name* (also called a *pathname* on some systems) tells the system how to locate an object.

Links in the QSYS.LIB file system

Symbolic links cannot be created or stored in the QSYS.LIB file system.

The relationship between a library and objects in a library is the equivalent of one hard link between the library and each object in the library. The integrated file system handles the library-object relationship as a link. Thus, it is possible to link from a file system that supports symbolic links to an object in the QSYS.LIB file system.

Related concepts

“Link” on page 11

A *link* is a named connection between a directory and an object. A user or a program can tell the system where to find an object by specifying the name of a link to the object. A link can be used as a path name or as part of a path name.

Integrated file system commands and displays in the QSYS.LIB file system

Many integrated file system commands and displays are valid in the QSYS.LIB file system.

The commands listed in “Accessing using CL commands” on page 65 can operate on the QSYS.LIB file system, except for the following restrictions:

- The ADDLNK command can be used only to create a symbolic link *to* an object in QSYS.LIB.
- File operations can be done only on program-described physical files and source physical files.
- The STRJRN and ENDJRN commands cannot be used on database physical files.
- The RCLLNK command is not supported.

The same restrictions apply to the user displays described in “Accessing using menus and displays” on page 64.

Integrated file system APIs in the QSYS.LIB file system

Many integrated file system APIs are valid in the QSYS.LIB file system.

The APIs listed in “Performing operations using APIs” on page 104 can operate on the QSYS.LIB file system, except for the following restrictions:

- File operations can be done only on program-described physical files and source physical files.
- The symlink() function can be used only to link *to* an object in QSYS.LIB from another file system that supports symbolic links.
- The QjoStartJournal() and QjoEndJournal() APIs cannot be used on database physical files.

Related information

Application programming interfaces (APIs)

Independent ASP QSYS.LIB

The independent ASP QSYS.LIB file system supports the i5/OS library structure in independent auxiliary storage pools (ASPs) you create and define. This file system provides access to database files and all of the other i5/OS object types that the library support manages in the independent ASPs.

In addition, it:

- Supports all user interfaces and programming interfaces that operate on i5/OS libraries and objects in those libraries in independent ASPs
- Supports all programming languages and facilities that operate on database files
- Provides extensive administrative support for managing i5/OS objects
- Supports stream I/O operations on physical file members, user spaces, and save files

Accessing independent ASP QSYS.LIB through the integrated file system interface

The independent ASP QSYS.LIB file system can be accessed through the integrated file system interface using either the i5/OS file server or the integrated file system commands, user displays, and APIs.

In using the integrated file system interfaces, you should be aware of some considerations and limitations.

QPWFSEVER authorization list in the independent ASP QSYS.LIB file system

The QPWFSEVER is an authorization list (object type *AUTL) that provides additional access requirements for all objects in the independent ASP QSYS.LIB file system being accessed through remote clients.

The authorities specified in this authorization list apply to all objects within the independent ASP QSYS.LIB file system.

The default authority to this object is PUBLIC *USE authority. The administrator can use the EDTAUTL (Edit Authorization List) or WRKAUTL (Work With Authorization List) commands to change the value of this authority. The administrator can assign PUBLIC *EXCLUDE authority to the authorization list so that the general public cannot access independent ASP QSYS.LIB objects from remote clients.

File handling restrictions in the independent ASP QSYS.LIB file system

Here are the restrictions to be aware of when handling files in the independent ASP QSYS.LIB file system

- Logical files are not supported.
- Physical files supported for text mode access are program-described physical files containing a single field and source physical files containing a single text field. Physical files supported for binary mode access include externally-described physical files in addition to those files supported for text mode access.
- Byte-range locking is not supported. For more information about byte-range locking, see the `fcntl()`--Perform File Control Command topic.
- If any job has a database file member open, only one job is given write access to that file member at any time. Other requests are allowed only read access.

Support for user spaces in the independent ASP QSYS.LIB file system

Independent ASP QSYS.LIB supports stream input and output operations to user space objects.

For example, a program can write stream data to a user space and read data from a user space. The maximum size of a user space is 16 776 704 bytes.

Be aware that user spaces are not tagged with a CCSID (coded character set identifier). Therefore, the CCSID returned is the default CCSID of the job.

Support for save files in the independent ASP QSYS.LIB file system

The independent ASP QSYS.LIB supports stream I/O operations to save file objects.

For example, an existing save file has data that may be read out or copied to another file until it is necessary to place the data into a different, existing, and empty save file object. When a save file is open for writing, no other open instances of the file are allowed. A save file **does** allow multiple open instances for reading, provided no job has more than one open instance of the file for reading. A save file may not be opened for read/write access. Stream I/O operations to save file data are not allowed when multiple threads are running in a job.

Stream I/O operations on a save file are not supported when the save file or its directory are being exported through the Network File System. They may, however, be accessed from PC clients and through the QFileSvr.400 file system.

Case-sensitivity in the independent ASP QSYS.LIB file system

In general, the independent ASP QSYS.LIB file system does not distinguish between uppercase and lowercase characters in the names of objects.

A search for object names achieves the same result regardless of whether characters in the names are uppercase or lowercase.

However, if a name is enclosed in quotation marks, the case of each character in the name is preserved. A search involving quoted names, therefore, is sensitive to the case of the characters in the quoted name.

Path names in the independent ASP QSYS.LIB file system

Each component of the path name must contain the object name followed by the object type of the object.

- For example:

```
/asp_name/QSYS.LIB/QGPL.LIB/PRT1.OUTQ
```

```
/asp_name/QSYS.LIB/EMP.LIB/PAY.FILE/TAX.MBR
```

where `asp_name` is the name of the independent ASP. The object name and object type are separated by a period (.). Objects in a library can have the same name if they are different object types, so the object type must be specified to uniquely identify the object.

- The object name in each component can be up to 10 characters long, and the object type can be up to 6 characters long.
- The directory hierarchy within independent ASP QSYS.LIB can be either two or three levels deep (two or three components in the path name), depending on the type of object being accessed. If the object is a database file, the hierarchy can contain three levels (library, file, member); otherwise, there can be only two levels (library, object). The combination of the length of each component name and the number of directory levels determines the maximum length of the path name.

If /, `asp_name`, and QSYS.LIB are included as the first three levels, the directory hierarchy for the Independent ASP QSYS.LIB file system can be up to six levels deep.

- The characters in names are converted to coded character set identifier (CCSID) 37 when the names are stored. Quoted names, however, are stored using the CCSID of the job.

For more information about CCSID, see the i5/OS globalization topic in the i5/OS Information Center.

Related concepts

“Path name” on page 14

A *path name* (also called a *pathname* on some systems) tells the system how to locate an object.

Links in the independent ASP QSYS.LIB file system

Symbolic links cannot be created or stored in the Independent ASP QSYS.LIB file system.

The relationship between a library and objects in a library is the equivalent of one hard link between the library and each object in the library. The integrated file system handles the library-object relationship as a link. Thus, it is possible to link from a file system that supports symbolic links to an object in the independent ASP QSYS.LIB file system.

Related concepts

“Link” on page 11

A *link* is a named connection between a directory and an object. A user or a program can tell the system where to find an object by specifying the name of a link to the object. A link can be used as a path name or as part of a path name.

Integrated file system commands and displays in the independent ASP QSYS.LIB file system

Many integrated file system commands and displays are valid in the independent ASP QSYS.LIB file system.

Nearly all the commands listed in “Accessing using CL commands” on page 65 can operate on the independent ASP QSYS.LIB file system. But there are a few exceptions:

- The ADDLNK command can be used only to create a symbolic link to an object in independent ASP QSYS.LIB.
- File operations can be done only on program-described physical files and source physical files.
- The STRJRN and ENDJRN commands cannot be used on database physical files.

- You cannot move libraries in the independent ASP QSYS.LIB file system to basic auxiliary storage pools (ASPs) using the MOV command. However, you can move libraries in independent ASP QSYS.LIB to the system ASP or other independent ASPs.
- If you use SAV or RST to save or restore library objects on an independent ASP, then that independent ASP must be associated with the job doing the SAV or RST operation, or the independent ASP must be specified on the ASPDEV parameter. The path name naming convention of /asp_name/QSYS.LIB/object.type is not supported on SAV and RST.
- The RCLLNK command is not supported.

The same restrictions apply to the user displays described in “Accessing using menus and displays” on page 64.

Integrated file system APIs in the independent ASP QSYS.LIB file system

Many integrated file system APIs are valid in the independent ASP QSYS.LIB file system

The APIs listed in “Performing operations using APIs” on page 104 can operate on the independent ASP QSYS.LIB file system, except for the following situations:

- File operations can be done only on program-described physical files and source physical files.
- The symlink() function can be used only to link to an object in independent ASP QSYS.LIB from another file system that supports symbolic links.
- The QjoStartJournal() and QjoEndJournal() APIs cannot be used on database physical files.
- If you use QsrSave() or QsrRestore() APIs to save or restore library objects on an independent ASP, this independent ASP must be associated with the job doing the save or restore operation, or the independent ASP must be specified on the ASPDEV key. The naming convention of path name (/asp_name/QSYS.LIB/object.type) is not supported on QsrSave() and QsrRestore() APIs.

Related information

Application programming interfaces (APIs)

Document library services file system (QDLS)

The QDLS file system supports the folders structure. It provides access to documents and folders.

In addition, it:

- Supports i5/OS folders and document library objects (DLOs).
- Supports data stored in stream files.

Accessing QDLS through the integrated file system interface

The QDLS file system can be accessed through the integrated file system interface using either the i5/OS file server or the integrated file system commands, user displays, and APIs.

In using the integrated file system interfaces, you should be aware of the following considerations and limitations.

Integrated file system and HFS in the QDLS file system

Operations can be performed on objects in the QDLS file system not only through the Document Library Objects (DLO) CL commands but also through either the integrated file system interface or APIs provided by a hierarchical file system (HFS).

Whereas the integrated file system is based on the Integrated Language Environment® (ILE) program model, HFS is based on the original System i program model.

The HFS APIs allow you to perform a few additional operations that the integrated file system does not support. In particular, you can use HFS APIs to access and change directory extended attributes (also

called *directory entry attributes*). Be aware that the naming rules for using HFS APIs are different from the naming rules for APIs using the integrated file system interface.

Related information

Hierarchical file system APIs

User enrollment in the QDLS file system

You must be enrolled in the system distribution directory when working with objects in the QDLS file system.

Case-sensitivity in the QDLS file system

The QDLS file system converts the lowercase English alphabetic characters **a** to **z** to uppercase when used in object names. Therefore, a search for object names using only those characters is not case-sensitive.

All other characters are case-sensitive in QDLS.

Related information

Folder and document name

Path names in the QDLS file system

Each component of the path name can consist of just a name.

- For example:

`/QDLS/FLR1/DOC1`

or a name plus an extension (similar to a DOS file extension), such as:

`/QDLS/FLR1/DOC1.TXT`

- The name in each component can be up to 8 characters long, and the extension (if any) can be up to 3 characters long. The maximum length of the path name is 82 characters, assuming an absolute path name that begins with `/QDLS`.
- The directory hierarchy within the document library services (QDLS) file system can be 32 levels deep. If `/` and `QDLS` are included as the first two levels, the directory hierarchy can be 34 levels deep.
- The characters in names are converted to the code page of the job when the names are stored unless data area `Q0DEC500` has been created in the `QUSRSYS` library. If this data area exists, then the characters in names are converted to code page 500 when the names are stored. This function provides compatibility with the behavior of the QDLS file system in previous releases. A name may be rejected if it cannot be converted to the appropriate code page.

For more information about code pages, see the `i5/OS globalization` topic in the `i5/OS Information Center`.

Related concepts

“Path name” on page 14

A *path name* (also called a *pathname* on some systems) tells the system how to locate an object.

Links in the QDLS file system

Symbolic links cannot be created or stored in the QDLS file system.

The integrated file system handles the relationship between a folder and document library objects in a folder as the equivalent of one link between the folder and each object in the folder. Thus, it is possible to link to an object in the QDLS file system from a file system that supports symbolic links.

Related concepts

“Link” on page 11

A *link* is a named connection between a directory and an object. A user or a program can tell the system where to find an object by specifying the name of a link to the object. A link can be used as a path name or as part of a path name.

Integrated file system commands and displays in the QDLS file system

Many integrated file system commands and displays are valid in the QDLS file system.

The commands listed in “Accessing using CL commands” on page 65 can operate on the QDLS file system, except for the following commands:

- The ADDLNK command can be used only to link *to* an object in QDLS from another file system that supports symbolic links.
- The CHKIN and CHKOUT commands are supported for files, but not for directories.
- These commands are not supported:
 - APYJRNCHG
 - CHGJRNOBJ
 - | – DSPJRN
 - ENDJRN
 - | – RCLLNK
 - | – RCVJRNE
 - | – RTVJRNE
 - SNDJRNE
 - STRJRN

The same restrictions apply to the user displays described in “Accessing using menus and displays” on page 64.

Integrated file system APIs in the QDLS file system

Many integrated file system APIs are valid in the QDLS file system.

The APIs listed in “Performing operations using APIs” on page 104 can operate on the QDLS file system, except for the following APIs:

- The symlink() function can be used only to link to an object in QDLS from another file system that supports symbolic links.
- The following functions are not supported:
 - givedescriptor()
 - ioctl()
 - link()
 - QjoEndJournal()
 - | – QjoRetrieveJournalEntries()
 - QjoRetrieveJournalInformation()
 - QJORJIDI()
 - QJOSJRNE()
 - QjoStartJournal()
 - Qp0lGetPathFromFileID()
 - readlink()
 - takedescriptor()

Related information

Application programming interfaces (APIs)

Optical file system (QOPT)

The QOPT file system provides access to stream data that is stored on optical media.

In addition, it:

- Provides a hierarchical directory structure similar to PC operating systems such as DOS and OS/2.
- Is optimized for stream file input and output.
- Supports data stored in stream files.

Accessing QOPT through the integrated file system

The QOPT file system can be accessed through the integrated file system using either the PC server or the integrated file system commands, user displays, and APIs.

In using the integrated file system interface, you should be aware of the following considerations and limitations.

Related information

Optical storage

Integrated file system and HFS in the QOPT file system

Operations can be performed on objects in the QOPT file system through either the integrated file system interface or APIs provided by a hierarchical file system (HFS).

Whereas the integrated file system is based on the Integrated Language Environment (ILE) program model, HFS is based on the original System i program model.

The HFS APIs allow you to perform a few additional operations that the integrated file system does not support. In particular, you can use HFS APIs to access and change directory extended attributes (also called *directory entry attributes*) or to work with held optical files. Be aware that the naming rules for using HFS APIs are different from the naming rules for APIs using the integrated file system interface.

For more information about HFS APIs, see the Optical device programming topic collection.

Related information

Hierarchical file system APIs

Case-sensitivity in the QOPT file system

Depending on the format of the optical media, case may or may not be preserved when creating files or directories in QOPT. However, file and directory searches are case-insensitive regardless of the optical media format.

Path names in the QOPT file system

The path name must begin with a slash (/). The path is made up of the file system name, the volume name, the directory and subdirectory names, and the file name.

- For example:

```
/QOPT/VOLUMENAME/DIRECTORYNAME/SUBDIRECTORYNAME/FILENAME
```

- The file system name, QOPT, is required.
- The volume and path name length vary by optical media format.
- You can specify /QOPT in the path name or include one or more directories or subdirectories in the path name. Directory and file names allow any character except X'00' through X'3F', X'FF'. Additional restrictions may apply based on the optical media format.
- The file name is the last element in the path name. The file name length is limited by the directory name length in the path.

For more details on path name rules in the QOPT file system, see the “Path Name Rules” discussion in Path names.

Related concepts

“Path name” on page 14

A *path name* (also called a *pathname* on some systems) tells the system how to locate an object.

Links in the QOPT file system

The QOPT file system supports only one link to an object. Symbolic links cannot be created or stored in QOPT.

However, files in QOPT can be accessed by using a symbolic link from the “root” (/) or QOpenSys file system.

Related concepts

“Link” on page 11

A *link* is a named connection between a directory and an object. A user or a program can tell the system where to find an object by specifying the name of a link to the object. A link can be used as a path name or as part of a path name.

Integrated file system commands and displays in the QOPT file system

Many integrated file system commands and displays are valid in the QOPT file system.

Most commands listed in “Accessing using CL commands” on page 65 can operate on the QOPT file system. There are, however, a few exceptions in the QOPT file system. Keep in mind that it may not be safe to use these CL commands in a multi-thread capable process; Certain restrictions may apply, depending on the optical media format. The same restrictions apply to the user displays described in “Accessing using menus and displays” on page 64.

The following integrated file system commands are not supported by the QOPT file system:

- ADDLNK
- APYJRNCHG
- CHGJRNOBJ
- CHKIN
- CHKOUT
- | • DSPJRN
- ENDJRN
- | • RCLLNK
- | • RCVJRNE
- | • RTVJRNE
- SNDJRNE
- STRJRN
- WRKOBJOWN
- WRKOBJPGP

Integrated file system APIs in the QOPT file system

Many integrated file system APIs are valid in the QOPT file system.

All of the APIs listed in “Performing operations using APIs” on page 104 can operate on the “root” (/) file system in a threadsafe manner, except for the following APIs:

- QjoEndJournal()
- | • QjoRetrieveJournalEntries()
- QjoRetrieveJournalInformation()

- QJORJIDI()
- QJOSJRNE()
- QjoStartJournal()

Related information

Application programming interfaces (APIs)

NetWare file system (QNetWare)

The QNetWare file system provides access to data on stand-alone PC servers that are running Novell NetWare 5.1 or 6.0.

In addition, QNetWare also provides the following functionality:

- Provides access to NetWare Directory Services (NDS) objects.
- Supports data stored in stream files.
- Provides dynamic mounting of NetWare file systems into the local name space

Notes:

1. The QNetWare file system is available only when NetWare Enhanced Integration, BOSS option 25, is installed on the system. After the next IPL following the installation, the /QNetWare directory and its subdirectories appear as part of the integrated file system directory structure.
2. The NetWare Enhanced Integration product does not support Novell Storage Services (NSS), so access to the data contained on that partition might be limited or restricted.

Mounting NetWare file systems

NetWare file systems located on Novell NetWare servers can be mounted on the “root” (/), QOpenSys, and other file systems to make access easier and perform better than under the /QNetWare directory.

Mounting NetWare file systems can also be used to take advantage of the options on the Add Mounted FS (ADDMFS) command, such as mounting a read-write file system as read-only. See the Add Mounted FS (ADDMFS) command topic for more information.

NetWare file systems can be mounted using an NetWare Directory Services (NDS) path or by specifying a NetWare path in the form of SERVER/VOLUME:directory/directory. For example, to mount the doorway located in volume Nest on server Dreyfuss, you can use this syntax:

```
DREYFUSS/NEST:doorway
```

This path syntax is very similar to the NetWare MAP command syntax. NDS paths can be used to specify a path to a NetWare volume but they cannot be mounted.

QNetWare directory structure

The /QNetWare directory structure represents multiple distinct file systems.

- The structure represents Novell NetWare servers and volumes out in the network in the following form:

```
/QNetWare/SERVER.SVR/VOLUME
```

The extension .SVR is used to represent a Novell NetWare server.

- When a volume under a server is accessed either through the integrated file system menus, commands, or APIs, the root directory of the NetWare volume is automatically mounted on the VOLUME directory under /QNetWare.
- QNetWare represents NDS trees on the network in the following form:

```
/QNetWare/CORP_TREE.TRE/USA.C/ORG.0/ORG_UNIT.OU/SVR1_VOL.CN
```

The extension .TRE is used to represent NDS trees, .C represents countries, .O represents organizations, .OU represents organizational units, and .CN is used to represent common names. If a Novell NetWare volume is accessed through the NDS path through a volume object or an alias to a volume object, its root directory is also automatically mounted on the NDS object.

Accessing QNetWare through the integrated file system interface

The QNetWare file system can be accessed through the integrated file system interface using either the i5/OS file server or the integrated file system commands, user displays, and APIs.

You should be aware of the following considerations, limitations, and dependencies.

Authorities and ownership in the QNetWare file system

Files and directories in QNetWare are stored and managed by Novell NetWare servers.

When using commands and APIs to retrieve or set the authorities of either owners or users, QNetWare maps NetWare users to System i users based on the name of a user. If the NetWare name exceeds 10 characters or a corresponding System i user does not exist, then the authority is not mapped. Owners that cannot be mapped are automatically mapped to the user profile QDFTOWN. The authorities of users can be displayed and changed using the Work with Authority (WRKAUT) and Change Authority (CHGAUT) commands. When authorities are transferred to and from the system, they are mapped to i5/OS authorities.

Audition in the QNetWare file system

Although Novell NetWare supports the auditing of files and directories, the QNetWare file system cannot change the auditing values of these objects. Therefore, the Change Auditing Value (CHGAUD) command is not supported.

Files and directories in the QNetWare file system

The QNetWare file system does not preserve the case in which files or directories are entered in a command or API.

All names are set to uppercase in transmission to the NetWare server. Novell NetWare also supports the namespaces of multiple platforms, such as DOS, OS/2, Apple Macintosh, and NFS. The QNetWare file system only supports the DOS namespace. Since the DOS namespace is required on all Novell NetWare volumes, all files and directories will appear in the QNetWare file system.

NetWare Directory Services objects in the QNetWare file system

The QNetWare file system supports the display of NetWare Directory Services (NDS) names in uppercase and lowercase.

Links in the QNetWare file system

The QNetWare file system supports only one link to an object. Symbolic links cannot be created or stored in QNetWare.

However, symbolic links can be created in the "root" (/) or QOpenSys directories that point to a QNetWare file or directory.

Related concepts

"Link" on page 11

A *link* is a named connection between a directory and an object. A user or a program can tell the system where to find an object by specifying the name of a link to the object. A link can be used as a path name or as part of a path name.

Integrated file system commands and displays in the QNetWare file system

Many integrated file system commands and displays are valid in the QNetWare file system

The commands listed in “Accessing using CL commands” on page 65 can operate on the QNetWare file system, except for the following commands:

- ADDLINK
- APYJRNCHG
- CHGAUD
- CHGJRNOBJ
- CHGPGP
- CHKIN
- CHKOUT
- | • DSPJRN
- ENDJRN
- | • RCLLNK
- | • RCVJRNE
- | • RTVJRNE
- SNDJRNE
- STRJRN
- WRKOBJOWN
- WRKOBJPGP

In addition to the previous commands, the following commands cannot be used against NDS objects, servers, or volumes:

- CHGOWN
- CPYFRMSTMF
- CPYTOSTMF
- CRTDIR

Integrated file system APIs in the QNetWare file system

Many integrated file system APIs are valid in the QNetWare file system.

The APIs listed in “Performing operations using APIs” on page 104 can operate on the QNetWare file system, except for the following APIs:

- givedescriptor()
- link()
- QjoEndJournal()
- | • QjoRetrieveJournalEntries()
- QjoRetrieveJournalInformation()
- QJORJIDI()
- QJOSJRNE()
- QjoStartJournal()
- readlink()
- symlink()
- takedescriptor()

In addition to the previous APIs, the following APIs cannot be used against NDS objects, servers, or volumes:

- chmod()
- chown()
- create()

- fchmod()
- fchown()
- fcntl()
- ftruncate()
- lseek()
- mkdir()
- read()
- readv()
- unmask()
- write()
- writev()

Related information

Application programming interfaces (APIs)

iSeries NetClient file system (QNTC)

The QNTC file system provides access to data and objects that are stored on an Integrated xSeries Server for iSeries running Windows NT 4.0 Server or later, or Linux. The QNTC file system also provides access to data and objects that are stored on remote servers running Windows NT 4.0 or later, Linux Samba 3.0 or later, or supported versions of iSeries NetServer.

The QNTC file system allows i5/OS applications to use data stored on Windows or Linux servers.

The QNTC file system is part of the base i5/OS operating system. It is not necessary to have the Integrated Server Support, option 29 of the operating system, installed to access /QNTC.

Accessing QNTC through the integrated file system interface

By using the iSeries NetServer, iSeries Navigator, integrated file system commands, user displays, or APIs, you can access the QNTC file system through the integrated file system interface.

Be aware of the following considerations and limitations.

Authorities and ownership in the QNTC file system

The QNTC file system does not support the ownership concept of a file or directory.

Attempts to use a command or API to change the ownership of files that are stored in QNTC will fail. A system user profile, called QDFTOWN, owns all of the files and directories in QNTC.

The authority to NT server files and directories is administered from the Windows NT server. QNTC does not support the WRKAUT and CHGAUT commands.

Case-sensitivity in the QNTC file system

The QNTC file system preserves the same uppercase and lowercase form in which object names are entered, but does not distinguish between uppercase and lowercase in the names.

A search for object names achieves the same result regardless of whether characters in the names are uppercase or lowercase.

Path names in the QNTC file system

The path consists of the file system name, the server name, the share name, the directory and subdirectory names, and the object name.

The requirements for a path name are as follows:

- The path name must begin with a slash and can be up to 255 characters long. Path names have the following form:

/QNTC/Servername/Sharename/Directory/ . . . /Object
(QNTC is a required part of the path name.)

- Path names are case sensitive.
- The server name can be up to 15 characters long. It must be part of the path.
- The share name can be up to 12 characters long.
- Each component of the path name after the share name can be up to 255 characters long.
- Within QNTC, 130 levels of hierarchy are generally available. If all components of the path name are included as hierarchy levels, the directory hierarchy can be as many as 132 levels deep.
- Names are stored in the Unicode CCSID.
- By default, each functional supported server in the local subnet will automatically appear as a directory under /QNTC. Use the Create Directory (MKDIR) command or mkdir() API to add accessible systems located outside the local subnet.

Related concepts

“Path name” on page 14

A *path name* (also called a *pathname* on some systems) tells the system how to locate an object.

Related information

Create Directory (MKDIR) command

mkdir()--Make Directory API

Links in the QNTC file system

The QNTC file system supports only one link to an object. You cannot create or store symbolic links in QNTC.

You can use a symbolic link from the “root” (/) or QOpenSys file system to access data in QNTC.

Related concepts

“Link” on page 11

A *link* is a named connection between a directory and an object. A user or a program can tell the system where to find an object by specifying the name of a link to the object. A link can be used as a path name or as part of a path name.

Integrated file system commands and displays in the QNTC file system

Many integrated file system commands and displays are valid in the QNTC file system

The commands listed in “Accessing using CL commands” on page 65 can operate on the QNTC file system, except for the following commands:

- ADDLNK
- APYJRNCHG
- CHGJRNOBJ
- CHGOWN
- CHGAUT
- CHGPGP
- CHKIN
- CHKOUT
- DSPAUT
- DSPJRN
- ENDJRN
- RCLLNK

- | • RCVJRNE
- | • RTVJRNE
- RST (available with Integrated xSeries Servers)
- SAV (available with Integrated xSeries Servers)
- SNDJRNE
- STRJRN
- WRKAUT
- WRKOBJOWN
- WRKOBJPGP

The same restrictions apply to the user displays that are described in “Accessing using menus and displays” on page 64.

| **QNTC environment variables**

| The network browsing behavior of QNTC can be controlled by two environment variables. Support for these environment variables began in i5/OS V5R4. Use the ADDENVVAR CL command to create these environment variables.

| **QZLC_SERVERLIST**

| When this environment variable is set to "2", all servers that appear in the /QNTC directory in the integrated file system can be accessed by QNTC. This was the default behavior before V5R4. When this variable is not set to "2" or has not been created, some servers that appear in the /QNTC directory might not be accessible.

| **QIBM_ZLC_NO_BROWSE**

| When this environment variable is set to "1", the /QNTC directory will only contain servers that were created with the MKDIR CL command or mkdir() API. The performance of many operations against the QNTC file system will improve when this environment variable has been set. But all /QNTC directories need to be created using the CL command.

Creating directories in the QNTC file system

You can use the Create Directory (MKDIR) command or mkdir() API to add a server directory to the /QNTC directory.

| By default, a QNTC directory is automatically created for all functional servers in the iSeries NetServer domain and the local subnet. Servers outside the local subnet or iSeries NetServer domain must be added using the MKDIR command or mkdir() API. For example:

```
MKDIR '/QNTC/NTSRV1'
```

adds the NTSRV1 server into the QNTC file system directory structure to enable access of files and directories on that server.

You can also add the a new server to the directory structure by using the TCP/IP address. For example:

```
MKDIR '/QNTC/9.130.67.24'
```

adds the server into the QNTC file system directory structure.

Notes:

- By configuring iSeries NetServer for WINS, it is possible to automatically create directories for servers beyond your subnet.

- If you use mkdir() API or the MKDIR CL command to add directories to the directory structure, they will not remain visible across IPLs. The MKDIR command or mkdir() API must be reissued after every system IPL.

If you prefer to add directories using the API or CL command, you can improve the performance of these commands by adding the environment variable QIBM_ZLC_NO_BROWSE, as in the following example:

```
ADDENVVAR ENVVAR(QIBM_ZLC_NO_BROWSE) VALUE(1) LEVEL(*SYS)
```

This environment variable causes the file system to bypass all network browsing when performing file operations.

Related information

Create Directory (MKDIR) command

mkdir()--Make Directory API

Integrated file system APIs in the QNTC file system

Many integrated file system APIs are valid in the QNTC file system.

The APIs listed in “Performing operations using APIs” on page 104 can operate on the QNTC file system, except for the following APIs:

- The chmod(), fchmod(), utime(), and umask() functions will have no effect on objects in QNTC, but attempting to use them will not cause an error.

- The QNTC file system does not support the following functions:

- chown()
- fchown()
- fclear()
- fclear64()
- givedescriptor()
- link()
- QjoEndJournal()
- QjoRetrieveJournalEntries()
- QjoRetrieveJournalInformation()
- QJORJIDI()
- QJOSJRNE()
- QjoStartJournal()
- Qp0lGetPathFromFileID()
- readlink()
- symlink()
- takedescriptor()

Related information

Application programming interfaces (APIs)

Enabling QNTC file system for Network Authentication Service

The QNTC file system enables System i platform access to Common Integrated File System (CIFS) servers that support the Kerberos V5 authentication protocol.

Rather than using a LAN manager type password to authenticate with each server, a properly configured System i platform will now be able to access supported CIFS servers with a single logon transaction.

To enable the Network Authentication Service (NAS) for use with QNTC, you must configure these items:

- Network Authentication Service (NAS)
- Enterprise Identity Mapping (EIM)

Once the above items have been configured, you can then enable a user to use NAS with the QNTC file system. The following steps are needed to allow a user to take advantage of the QNTC NAS support.

- The user's i5/OS user profile must have the local password management parameter, LCLPMDMGT, set to ***NO**. By specifying ***NO**, the user will not have a password to the server and will not be able to sign on to a 5250 session. The only access to the server will be through NAS-enabled applications, such as iSeries Navigator or iSeries Access 5250 Display Emulator.

If the user specifies ***YES**, the password will be managed by the server and the user will be authenticated without NAS.

- You must have a Kerberos ticket and an iSeries Navigator connection.
- The Kerberos ticket for the System i platform you are using must be forwardable. To make a ticket forwardable, follow these steps:
 1. Access the **Active Directory Users and Computers** tool on the KDC for your NAS realm.
 2. Select users.
 3. Select the name that corresponds to the service principal name.
 4. Select **Properties**.
 5. Select the **Account** tab.
 6. Select **Account is trusted for delegation**.

Related information

Network authentication service

Enterprise Identity Mapping (EIM)

i5/OS file server file system (QFileSvr.400)

The QFileSvr.400 file system provides transparent access to other file systems that reside on remote System i platforms. It is accessed through a hierarchical directory structure.

The QFileSvr.400 file system can be thought of as a client that acts on behalf of users to perform file requests. QFileSvr.400 interacts with the i5/OS file server on the target system to perform the actual file operation.

Accessing QFileSvr.400 through the integrated file system interface

The QFileSvr.400 file system can be accessed through the integrated file system interface using either the i5/OS file server or the integrated file system commands, user displays, and APIs.

In using the integrated file system interfaces, you should be aware of the following considerations and limitations.

Note: The characteristics of the QFileSvr.400 file system are determined by the characteristics of the file system being accessed on the target server.

Case-sensitivity in the QFileSvr.400 file system

For a first-level directory, which actually represents the "root" (/) directory of the target system, the QFileSvr.400 file system preserves the same uppercase and lowercase form in which object names are entered.

However, no distinction is made between uppercase and lowercase when QFileSvr.400 searches for names.

For all other directories, case-sensitivity is dependent on the specific file system being accessed. QFileSvr.400 preserves the same uppercase and lowercase form in which object names are entered when file requests are sent to the i5/OS file server.

Path names in the QFileSvr.400 file system

Path names have specific form in the QFileSvr.400 file system.

- The form is:

```
/QFileSvr.400/RemoteLocationName/Directory/Directory . . . /Object
```

The first-level directory (that is, RemoteLocationName in the example shown above) represents both of the following attributes:

- The name of the target system that will be used to establish a communications connection. The target system name can be either of the following names:
 - A TCP/IP host name (for example, beowulf.newyork.corp.com)
 - An SNA LU 6.2 name (for example, appn.newyork)
- The “root” (/) directory of the target system

Therefore, when a first-level directory is created using an integrated file system interface, any specified attributes are ignored.

Note: First-level directories are not persistent across IPLs. That is, the first-level directories must be created again after each IPL.

- Each component of the path name can be up to 255 characters long. The full path name can be up to 16 megabytes long.

Note: The file system in which the object resides may restrict the component length and path name length to less than the maximum allowed by QFileSvr.400.

- There is no limit to the depth of the directory hierarchy, other than program and system limits and any limits imposed by the file system being accessed.
- The characters in names are converted to UCS2 Level 1 form when the names are stored.

Related concepts

“Name continuity” on page 17


When you use the “root” (/), QOpenSys, and user-defined file systems, you can take advantage of system support that ensures characters in object names remain the same.

“Path name” on page 14

A *path name* (also called a *pathname* on some systems) tells the system how to locate an object.

Communications in the QFileSvr.400 file system

The QFileSvr.400 file system communicates in the following ways.

- TCP connections with the file server on a target server can be established only if the QSERVER subsystem on the target server is active.
- SNA LU 6.2 connections are attempted only if there is a locally controlled session that is not in use (for example, a session specifically established for use by the LU 6.2 connection). When establishing LU 6.2 connections, the QFileSvr.400 file system uses a mode of BLANK. On the target system, a job named QPWFSERV is submitted to the QSERVER subsystem. The user profile of this job is defined by the communications entry for the BLANK mode. For more information about LU 6.2 communications, see [APPC Programming](#) .
- File server requests that use TCP as the communications protocol are performed within the context of the job that is issuing the request. File server requests that use SNA as the communications protocol are performed by the i5/OS system job Q400FILSVR.
- If a connection is not yet established with the target server, the QFileSvr.400 file system assumes that the first-level directory represents a TCP/IP host name. The QFileSvr.400 file system goes through the following steps to establish a connection with the target server:

1. Resolve the remote location name to an IP address.
2. Connect to the host server's server mapper on well-known port 449 using the resolved IP address. Then send a query to the server mapper for the service name "as-file." One of the following occurs as result of the query:
 - If "as-file" is in the service table on the target server, the server mapper returns the port on which the i5/OS file server daemon is listening.
 - If the server mapper is not active on the target server, the default port number for "as-file" (8473) is used.

The QFileSvr.400 file system then tries to establish a TCP connection with the i5/OS file server daemon on the target server. When the connection is established, QFileSvr.400 exchanges requests and replies with the file server. Within the QSERVER subsystem, the QPWFSERVSO prestart requests take control of the connection. Each prestart job runs under its own user profile.

3. If the remote location name is not resolved to an IP address, the first-level directory is assumed to be an SNA LU 6.2 name. Therefore, an attempt is made to establish an APPC connection with the i5/OS file server.
- The QFileSvr.400 file system periodically (every 2 hours) checks to determine if there are any connections that are not being used (for example, no opened files associated with the connection) and those connections had no activity during a 2-hour period. If such a connection is found, the connection is ended.
 - The QFileSvr.400 file system cannot detect loops. The following path name is an example of a loop:

/QFileSvr.400/Remote2/QFileSvr.400/Remote1/QFileSvr.400/Remote2/...

where Remote1 is the local system. When the path name that contains a loop is specified, the QFileSvr.400 file system returns an error after a brief period of time. The error indicates that a time-out has occurred.

The QFileSvr.400 file system will use an existing free session when communicating over SNA. It is necessary to start the mode and establish a session for the QFileSvr.400 to successfully connect to the remote communications system.

Security and object authority in the QFileSvr.400 file system

If both of the systems have Network Authentication Service and Enterprise Identity Mapping (EIM) configured, and the user has authenticated with Kerberos, then Kerberos can be used to authenticate to access a file system that resides on a target System i platform.

If the Kerberos authentication fails, then the user ID and password can be used to verify access.

Note: If the ticket-granting ticket or the System i ticket expires after the target system has verified your access, the expiration will not be effective until the connection to the target system has ended.

- To access a file system that resides on a target System i platform, you must have a user ID and password on the target system that matches the user ID and password on the local system if Kerberos is not used to authenticate.

Note: If your password on the local or target system is changed after the target system has verified your access, then the change is not reflected until the connection to the target system has ended. However, there is no delay if your user profile on the local system is deleted and another user profile is created with the same user ID. In this case, the QFileSvr.400 file system verifies that you have access to the target system.

- Object authority is based on the user profile that resides on the target system. That is, you are allowed to access an object in the file system on the target system only if your user profile on the target system has the appropriate authority to the object.

Related information

Network authentication service

Enterprise Identity Mapping (EIM)

Links in the QFileSvr.400 file system

The QFileSvr.400 file system supports only one link to an object.

Symbolic links cannot be created or stored in QFileSvr.400. However, files in QFileSvr.400 can be accessed by using a symbolic link from the “root” (/), QOpenSys, or user-defined file systems.

Related concepts

“Link” on page 11

A *link* is a named connection between a directory and an object. A user or a program can tell the system where to find an object by specifying the name of a link to the object. A link can be used as a path name or as part of a path name.

Integrated file system commands and displays in the QFileSvr.400 file system

Many integrated file system commands and displays are valid in the QFileSvr.400 file system.

The commands listed in “Accessing using CL commands” on page 65 can operate on the QFileSvr.400 file system, except for the following commands:

- ADDLNK
- APYJRNCHG
- CHGAUT
- CHGJRNOBJ
- CHGOWN
- DSPAUT
- | • DSPJRN
- ENDJRN
- | • RCLLNK
- | • RCVJRNE
- RST
- | • RTVJRNE
- SAV
- SNDJRNE
- STRJRN
- WRKOBJOWN
- WRKOBJPGP

The same restrictions apply to the user displays described in “Accessing using menus and displays” on page 64.

Integrated file system APIs in the QFileSvr.400 file system

Many integrated file system APIs are valid in the QFileSvr.400 file system.

The APIs listed in “Performing operations using APIs” on page 104 can operate on the QFileSvr.400 file system, except for the following APIs:

- chown()
- fchown()
- fclear()
- fclear64()
- givedescriptor()
- link()
- QjoEndJournal()

- l • QjoRetrieveJournalEntries()
- QjoRetrieveJournalInformation()
- QJORJIDI()
- QJOSJRNE
- QjoStartJournal
- Qp0lGetPathFromFileID()
- symlink()
- takedescriptor()

Related information

Application programming interfaces (APIs)

Network File System (NFS)

The Network File System (NFS) provides the user with access to data and objects that are stored on a remote NFS server.

An NFS server can export a Network File System that NFS clients will then mount dynamically.

In addition, any file system mounted locally through the Network File System will have the features, characteristics, limitations, and dependencies of the directory or file system it was mounted from on the remote server. Operations on mounted file systems are not performed locally. Requests flow through the connection to the server and must obey the requirements and restrictions of the type of file system on the server.

Accessing NFS file systems through the integrated file system interface

The NFS is accessible through the integrated file system interface. Be aware of these considerations and limitations.

Characteristics of the Network File System

The characteristics of any file system mounted through NFS are dependent on the type of file system that was mounted from the server.

It is important to realize that requests performed on what appears to be a local directory or file system are really operating on the server through the NFS connection.

This client/server relationship can be confusing. Consider, for example, that you mounted the QDLS file system from the server on top of a branch of the “root” (/) directory of your client. Although the mounted file system appears to be an extension of the local directory, it actually functions and performs as the QDLS file system.

Realizing this relationship for file systems mounted through NFS is important for processing requests locally and through the server connection. Just because a command processes correctly on the local level does not mean that it will work on the directory mounted from the server. Each directory mounted on the client will have the properties and characteristics of the server file system.

Variations of servers and clients in the Network File System

There are three major possibilities for client/server connections that can affect how the Network File System (NFS) will function and what its characteristics will be.

The possibilities are:

- The user mounts a file system from a System i platform on a client.
- The user mounts a file system from a UNIX platform on a client.

- The user mounts a file system on a client from a system that is neither a System i platform nor a UNIX platform.

In the first scenario, the mounted file system behaves on the client similarly to how it behaves on the System i platform. However, both the characteristics of the Network File System and the file system being served need to be taken into account. For example, if you mount the QDLS file system from the server to the client, it has the characteristics and limitations of the QDLS file system. For instance, in the QDLS file system, path name components are limited to 8 characters plus a 3-character extension. However, the mounted file system also has NFS characteristics and limitations. For example, you cannot use the CHGAUD command to change the auditing value of an NFS object.

In the second scenario, it is important to realize that any file system mounted from a UNIX server behaves most similarly to the i5/OS QOpenSys file system.

In the third scenario, you will need to review the documentation for the file system associated with the operating system.

Related reference

“Open systems file system (QOpenSys)” on page 31

The QOpenSys file system is compatible with open system standards based on UNIX, such as POSIX and X/Open Portability Guide (XPG). Like the “root” (/) file system, this file system takes advantage of the stream file and directory support that is provided by the integrated file system.

Links in the Network File System

Generally, multiple hard links to the same object are allowed in the Network File System.

Symbolic links are fully supported. A symbolic link can be used to link from a Network File System to an object in another file system. The capabilities for multiple hard links and symbolic links are completely dependent on the file system that is being mounted with NFS.

Related concepts

“Link” on page 11

A *link* is a named connection between a directory and an object. A user or a program can tell the system where to find an object by specifying the name of a link to the object. A link can be used as a path name or as part of a path name.

Integrated file system commands in the Network File System

Many integrated file system commands are valid in the Network File System (NFS).

All of the commands that are listed in “Accessing using CL commands” on page 65 and the displays that are described in “Accessing using menus and displays” on page 64 can operate on the Network File System, except for the following commands:

- APYJRNCHG
- CHGJRNOBJ
- CHGAUD
- CHGATR
- CHGAUT
- CHGOWN
- CHGPGP
- CHKIN
- CHKOUT
- | • DSPJRN
- ENDJRN
- | • RCLLNK

- | • RCVJRNE
- | • RTVJRNE
- SNDJRNE
- STRJRN

There are some CL commands that are specific to the Network File System and other mounted file systems in general. However, it may not be safe to use these commands in a multi-thread capable process. The following table describes these commands. For a complete description of commands and displays that are related specifically to the Network File System, see OS/400 Network File System

Support  .

Table 6. Network file system CL commands

Command	Description
ADDMFS	Add Mounted File System. Places exported, remote server file systems over local client directories.
CHGNFSEXP	Change Network File System. Export Adds or removes directory trees to the export table of file systems that are exported to Network File System clients.
DSPMFSINF	Display Mounted File System Information. Displays information about a mounted file system.
ENDNFSSVR	End Network File System Server. Ends one or all of the Network File System daemons on the server.
EXPORTFS	Export a File System. Adds or removes directory trees to the export table of file systems that are exported to Network File System clients.
MOUNT	Mount a File System. Places exported, remote server file systems over local client directories. This command is an alias for the ADDMFS command.
RLSIFSLCK	Release Integrated File System Locks. Releases all Network File System byte-range locks held by a client or on an object.
RMVMFS	Remove Mounted File System. Removes exported, remote server file systems from the local client namespace.
STRNFSSVR	Start Network File System Server. Starts one or all of the Network File System daemons on the server.
UNMOUNT	Unmount a File System. Removes exported, remote server file systems from the local client namespace. This command is an alias for the RMVMFS command.

Note: A Network File System must be mounted before any commands can be used on it.

Integrated file system APIs in the Network File System

Many integrated file system APIs are valid in the Network File System (NFS).

All of the APIs that are listed in “Performing operations using APIs” on page 104 can operate on the network file system, except for the following APIs:

- QjoEndJournal()
- | • QjoRetrieveJournalEntries()
- QjoRetrieveJournalInformation()
- QJORJIDI()
- QJOSJRNE()
- QjoStartJournal()

For a complete description of the C language functions that are related specifically to the Network File System, see OS/400 Network File System Support .

Note: A Network File System must be mounted before any APIs can be used on it.

Related information

Application programming interfaces (APIs)

Accessing the integrated file system

All of the user interfaces, such as menus, commands, and displays, that are used to work with your system's libraries, objects, database files, folders, and documents still operate as they did before the introduction of the integrated file system.

These interfaces, however, cannot be used to work with the stream files, directories, and other objects supported by the integrated file system.

A separate set of user interfaces is provided for the integrated file system. These interfaces can be used on objects in any file system that can be accessed through the integrated file system directories.

You can interact with the directories and objects of the integrated file system from your system by using menus and displays or by using control language (CL) commands. Additionally, you can use application programming interfaces (APIs) to take advantage of the stream files, directories, and other support of the integrated file system.

You can also interact with the integrated file system through iSeries Navigator, a graphical user interface (GUI) used for managing and administering your system from your Windows desktop.

Accessing using menus and displays

You can perform operations on files and other objects in the integrated file system by using a set of menus and displays provided by your system.

About this task

To display integrated file system menus:

1. Sign on to your system.
2. Press Enter to continue.
3. From the main menu, select the **Files, Libraries, and Folders** option.
4. From the Files, Libraries, and Folders menu, select the **Integrated File System** option.

Results

From here, you can work with Directory commands, Object commands, or Security commands in the integrated file system, depending on your needs. However, if you know the CL command you will be using, you can type it at the command line at the bottom of the screen and press **Enter**, bypassing the menu of options.

Additionally, you can access the integrated file system from any menu on your system by performing the following steps:

1. Type GO DATA on any command line to display the Files, Libraries, and Folders menu.
2. Select **Integrated file system**.

To see a menu of Network File System commands, type GO CMDNFS on any command line. To see a menu of user-defined file system commands, type GO CMDUDFS on any command line.

From the integrated file system menus, you can request displays or commands on which you can do the following operations:

- Create, convert, and remove a directory
- Display and change the name of the current directory
- Add, display, change, and remove object links
- Copy, move, and rename objects
- Check out and check in objects
- Save (back up) and restore objects
- Display and change object owners and user authorities
- Display and change attributes of objects
- Copy data between stream files and database file members
- Create, delete, and display the status of user-defined file systems
- Export file systems from a server
- Mount and unmount file systems on a client

Some file systems do not support all of these operations.

Related concepts

“File systems” on page 24

A *file system* provides you with the support to access specific segments of storage that are organized as logical units. These logical units on your system are files, directories, libraries, and objects.

Related reference

“Path name rules for CL commands and displays” on page 68

When using an integrated file system command or display to operate on an object, you identify the object by supplying its path name.

“Accessing using CL commands”

All of the operations that you can do through the integrated file system menus and displays can be done by entering control language (CL) commands. These commands can operate on files and other objects in any file system that are accessible through the integrated file system interface.

Accessing using CL commands

All of the operations that you can do through the integrated file system menus and displays can be done by entering control language (CL) commands. These commands can operate on files and other objects in any file system that are accessible through the integrated file system interface.

Table 1 summarizes the integrated file system commands. For more information about CL commands that are specifically related to user-defined file systems, the Network File System, and mounted file systems in general, see “User-defined file systems (UDFSs)” on page 33 and “Network File System (NFS)” on page 61. Where a command performs the same operation as an OS/2 or DOS command, an alias (an alternative command name) is provided for the convenience of OS/2 and DOS users.

Table 7. Integrated file system commands

Command	Description	Alias
ADDLNK	Add Link. Adds a link between a directory and an object.	
ADDMFS	Add Mounted File System. Places exported, remote server file systems over local client directories.	MOUNT
APYJRNCHG ²	Apply Journalled Changes. Uses journal entries to apply changes that have occurred since a journalled object was saved or to apply changes up to a specified point.	

Table 7. Integrated file system commands (continued)

Command	Description	Alias
CHGATR	Change Attribute. Changes an attribute for a single object, a group of objects, or a directory tree where the directory, its contents, and the contents of all of its subdirectories have the attribute changed.	
CHGAUD	Change Auditing Value. Turns auditing on or off for an object.	
CHGAUT	Change Authority. Gives specific authority for an object to a user or group of users.	
CHGCURDIR	Change Current Directory. Changes the directory to be used as the current directory.	CD, CHDIR
CHGJRNOBJ ²	Change Journalled Objects. Changes the journaling attributes of an object or list of objects without the need to end and restart journaling for the object.	
CHGNFSEXP	Change Network File System Export. Adds directory trees to or removes them from the export table that is exported to NFS clients.	EXPORTFS
CHGOWN	Change Owner. Transfers object ownership from one user to another.	
CHGPGP	Change Primary Group. Changes the primary group from one user to another.	
CHKIN	Check In. Checks in an object that was previously checked out.	
CHKOUT	Check Out. Checks out an object, which prevents other users from changing it.	
CPY	Copy. Copies a single object or a group of objects.	COPY
CPYFRMSTMF	Copy from Stream File. Copies data from a stream file to a database file member.	
CPYTOSTMF	Copy to Stream File. Copies data from a database file member to a stream file.	
CRTDIR	Create Directory. Adds a new directory to the system.	MD, MKDIR
CRTUDFS	Create UDFS. Creates a User-Defined File System.	
CVTDIR	Convert directory. Provides information about converting integrated file system directories from *TYPE1 format to *TYPE2 format.	
CVTRPCSRC	Convert RPC Source. Generates C code from an input file written in the Remote Procedure Call (RPC) language.	RPCGEN
DLTUDFS	Delete UDFS. Deletes a User-Defined File.	
DSPAUT	Display Authority. Shows a list of authorized users of an object and their authorities for the object.	
DSPCURDIR	Display Current Directory. Shows the name of the current directory.	
DSPJRN ²	Display Journal. Converts journal entries (contained in one or more receivers) into a form suitable for external representation.	
DSPLNK	Display Object Links. Shows a list of objects in a directory and provides options to display information about the objects.	
DSPF	Display Stream File. Displays a stream file or a database file.	
DSPMFSINF	Display Mounted File System Information. Displays information about a mounted file system.	STATFS
DSPUDFS	Display UDFS. Displays User-Defined File System.	

Table 7. Integrated file system commands (continued)

Command	Description	Alias
EDTF	Edit Stream File. Edits a stream file or a database file.	
ENDJRN ²	End Journal. Ends the journaling of changes for an object or list of objects.	
ENDNFSSVR	End Network File System Server. Ends one or all of the NFS daemons on the server and the client.	
ENDRPCBIND	End RPC Binder Daemon. Ends the Remote Procedure Call (RPC) RPCBind daemon.	
MOV	Move. Moves an object to a different directory	MOVE
PRDIRINF	Print Directory Information. Used to print directory information for objects in the integrated file system that was collected by the Retrieve Directory Information (RTVDIRINF) command.	
RCLLNK	Reclaim Objects Links. Identifies and, if possible, corrects problems in mounted file systems that are in use.	
RCVJRNE ²	Receive Journal Entry. Allows a specified user exit program to continuously receive journal entries.	
RLSIFSLCK	Release Integrated File System Locks. Releases all byte-range locks held by an NFS client or on an object.	
RMVDIR	Remove Directory. Removes a directory from the system.	RD, RMDIR
RMVLNK	Remove Link. Removes the link to an object	DEL, ERASE
RMVMFS	Remove Mounted File System. Removes exported, remote server file systems from the local client directories.	UNMOUNT
RNM	Rename. Changes the name of an object in a directory.	REN
RPCBIND	Start RPC Binder Daemon. Starts the Remote Procedure Call (RPC) RPCBind Daemon.	
RST	Restore. Copies an object or group of objects from a backup device to the system	
RTVCURDIR	Retrieve Current Directory. Retrieves the name of the current directory and puts it into a specified variable (used in CL programs)	
RTVDIRINF	Retrieve Directory Information. Collects attributes for objects in the integrated file system.	
RTVJRNE ²	Retrieve Journal Entry. Gets a particular journal entry and place the results in CL variables.	
SAV	Save. Copies an object or group of objects from the system to a backup device	
SNDJRNE ²	Send Journal Entry. Adds user journal entries, optionally associated with a journaled object, to a journal receiver.	
STRJRN ²	Start Journal. Starts journaling changes (made to an object or list of objects) to a specific journal.	
STRNFSSVR	Start Network File System Server. Starts one or all of the NFS daemons on the server and client.	
WRKAUT	Work with Authority. Shows a list of users and their authorities and provides options for adding a user, changing a user authority, or removing a user.	
WRKLNK	Work with Object Links. Shows a list of objects in a directory and provides options for performing actions on the objects.	

Table 7. Integrated file system commands (continued)

Command	Description	Alias
WRKOBJOWN ¹	Work with Objects by Owner. Shows a list of objects owned by a user profile and provides options for performing actions on the objects.	
WRKOBJPGP ¹	Work with Objects by Primary Group. Shows a list of objects controlled by a primary group and provides options for performing actions on the objects.	

Notes:

1. The WRKOBJOWN and WRKOBJPGP commands can display all object types but may not be fully functional in all file systems.
2. See Journal management in the i5/OS Information Center for more information.

Related concepts

“File systems” on page 24

A *file system* provides you with the support to access specific segments of storage that are organized as logical units. These logical units on your system are files, directories, libraries, and objects.

Related tasks

“Accessing using menus and displays” on page 64

You can perform operations on files and other objects in the integrated file system by using a set of menus and displays provided by your system.

Related information

Control language (CL)

Path name rules for CL commands and displays

When using an integrated file system command or display to operate on an object, you identify the object by supplying its path name.

The following list is a summary of rules to keep in mind when specifying path names. The term *object* in these rules refers to any directory, file, link, or other object:

- Object names must be unique within each directory.
- The path name that is passed to an integrated file system CL command must be represented in the coded character set identifier (CCSID) currently in effect for the job. If the CCSID of the job is 65535, the path name must be represented in the default CCSID of the job. Because text strings are normally encoded in CCSID 37, it is necessary to convert hard-coded path names to the job CCSID before passing the path to the command.
- Path names must be enclosed in single quotation marks (') when entered on a command line. These marks are optional when path names are entered on displays. If the path name includes any quoted strings, however, the enclosing ' ' marks must also be included.
- Path names are entered left to right, beginning with the highest-level directory and ending with the name of the object to be operated on by the command. The name of each component in the path is separated by a slash (/).

Note: Some CL commands also allow the backslash (\) to be used as a separator by automatically converting the backslash (\) to a slash (/). Some other CL commands, however, treat the backslash (\) no differently than they treat any other character. Therefore, the backslash (\) separator should be used with caution.

For example:

'Dir1/Dir2/Dir3/UsrFile'

or

'Dir1\Dir2\Dir3\UsrFile'

- The slash (/) and backslash (\) characters and nulls cannot be used in the individual components of the path name when the slash (/) and backslash (\) are used as separators. Lowercase letters are not changed to uppercase letters by the commands. The name might or might not be changed to uppercase, depending on whether the file system containing the object is case-sensitive and whether the object is being created or searched for.
- The length of an object name is limited by the file system the object is in and the maximum length of a command string. The commands accept object names up to 255 characters long and path names up to 5000 characters long.

| • A separator character (for example: /) at the beginning of a path name means that the path begins at the topmost directory, the “root” (/) directory; for example:

| '/Dir1/Dir2/Dir3/UsrFile'

| • If the path name does not begin with a separator character (for example: /) , the path is assumed to begin at the current directory of the user entering the command; for example:

| 'MyDir/MyFile'

| where MyDir is a subdirectory of the user’s current directory.

| • A tilde (~) character followed by a separator character (for example: /) at the beginning of a path name means that the path begins at the home directory of the user entering the command; for example:

| '~/UsrDir/UsrObj'

| • A tilde (~) character followed by a user name and then a separator character (for example: /) at the beginning of a path name means that the path begins at the home directory of the user identified by the user name; for example:

| '~user-name/UsrDir/UsrObj'

- In some commands, an asterisk (*) or a question mark (?) can be used in the last component of a path name to search for patterns of names. The * tells the system to search for names that have any number of characters in the position of the * character. The ? tells the system to search for names that have a single character in the position of the ? character. The following example searches for all objects whose names begin with *d* and end with *txt*:

 '/Dir1/Dir2/Dir3/d*txt'

The following example searches for objects whose names begin with *d* followed by any single character and end with *txt*:

 '/Dir1/Dir2/Dir3/d?txt'

- To avoid confusion with i5/OS special values, path names cannot start with a single asterisk (*) character. To perform a pattern match at the beginning of a path name, use two asterisks (**); for example:

 '**.file'

Note: This only applies to relative path names where no other characters precede the asterisk (*).

- When operating on objects in the QSYS.LIB file system, the component names must be of the form *name.object-type*; for example:

 '/QSYS.LIB/PAY.LIB/TAX.FILE'

- When operating on objects in the independent ASP QSYS.LIB file system, the component names must be of the form *name.object-type*; for example:

 '/asp_name/QSYS.LIB/PAYDAVE.LIB/PAY.FILE'

- The path name must be enclosed in additional sets of single quotation marks (') or quotation marks (") if any of the following characters is used in a component name:

– Asterisk (*)

Note: To avoid confusion with i5/OS special values, path names should not start with a single asterisk (*) character.

– Question mark (?)

– Single quotation mark (')

- Quotation mark (")
- Tilde (~), if used as the first character in the first component name of the path name (if used in any other position, the tilde is interpreted as just another character)

For example:

```
"/Dir1/Dir/A*Smith"
```

or

```
'''/Dir1/Dir/A*Smith'''
```

This practice is not recommended because the meaning of the character in a command string can be confusing and it is more likely that the command string might be entered incorrectly.

- Do not use a colon (:) in path names. It has a special meaning within the system.
- The processing support for commands and associated user displays does not recognize code points below hexadecimal 40 as characters that can be used in command strings or on displays. If these code points are used, they must be entered as a hexadecimal representation, such as the following example:

```
crtmdir dir(x'02')
```

Therefore, use of code points below hexadecimal 40 in path names is not recommended. This restriction applies only to commands and associated displays, not to APIs. In addition, a value of hexadecimal 0 is not allowed in path names.

Related concepts

“File systems” on page 24

A *file system* provides you with the support to access specific segments of storage that are organized as logical units. These logical units on your system are files, directories, libraries, and objects.

Related information

Control language (CL)

Working with output of the RTVDIRINF and PRTDIRINF commands

The Retrieve Directory Information (RTVDIRINF) command is used to collect attributes for objects in the integrated file system. The collected information is stored in database files (tables) that are named using the information file prefix specified by the INFFILEPFX parameter. The tables are created in the library specified by the INFLIB parameter.

Three tables are created as a result of the RTVDIRINF command. One table stores object attributes, one is for directories, and the last table is used to determine which files were used to store object attributes.

The following table describes the fields that are provided for the table that stores the object attributes. If *GEN is specified in the information file prefix (INFFILEPFX) parameter, the database files are created with a unique prefix generated by this command. The prefix begins with QAEZD followed by four digits. The files created to store the collected information are named using this prefix followed by either the letter D (for the file that contains directory information) or the letter O (for the file that contains information about objects in directories). For example, the first time that the command is run with *GEN specified, files QAEZD0001D and QAEZD0001O are created in the library specified by the Information library (INFLIB) parameter. Users can specify a file prefix to use for naming this database, which can be up to nine characters long.

Table 8. QAEZDxxxxO (store object's attributes)

Field name	Field type	Field description
QEZACCTIM	TIMESTAMP	The date and time that the object's data was last accessed.
QEZALCSIZE ¹	BIGINT	The number of bytes that have been allocated for this object.

Table 8. QAEZDxxxxO (store object's attributes) (continued)

Field name	Field type	Field description
QEZALWCKPW	SMALLINT	Whether a stream file (*STMF) can be shared with readers and writers during the save-while-active checkpoint processing. Valid values are: 0 - The object can be shared with readers only. 1 - The object can be shared with readers and writers.
QEZASP	SMALLINT	The auxiliary storage pool in which the object is stored.
QEZAUDT	GRAPHIC (10)	The auditing value associated with the object. Valid values are: *NONE - No auditing occurs for this object when it is read or changed regardless of the user who is accessing the object. *USRPRF - Audit this object only if the current user is being audited. The current user is tested to determine if auditing should be done for this object. The user profile can specify if only change access is audited or if both read and change accesses are audited for this object. *CHANGE - Audit all change access to this object by all users on the system. *ALL - Audit all access to this object by all users on the system. All access is defined as a read or change operation.
QEZAUTLST	GRAPHIC (10)	The name of the authorization list that is used to secure the named object. The value *NONE indicates that no authorization list is used in determining authority to the object.
QEZBLKSIZ	INTEGER	The block size of an object.
QEZCASE	SMALLINT	Indicates the Case Sensitivity of the file system that contains this object. 0 - File system is not case sensitive. 1 - File system is case sensitive.
QEZCCSID	INTEGER	The CCSID of the data and extended attributes of the object.
QEZCEAS	BIGINT	Number of critical extended attributes associated with this object.
QEZCHGTIMA ¹	TIMESTAMP	The date and time that the object's attributes were last changed.
QEZCHGTIMD	TIMESTAMP	The date and time that the object's data was last changed.
QEZCHKOUT ¹	SMALLINT	An indicator as to whether an object is checked out. Valid values are: 0 - The object is not checked out. 1 - The object is checked out.
QEZCHKOWN	GRAPHIC (10)	The user who has the object checked out. This field is blank if it is not checked out.
QEZCHKTIM	TIMESTAMP	The date and time the object was checked out. If the object is not checked out, this field will have NULL as its value.
QEZCLSTRSP	SMALLINT	The object is the storage that was allocated for Integrated xSeries servers to use as virtual disk drives for the IBM System x servers. From the perspective of the System i platform, virtual drives appear as byte stream files within the integrated file system. 0 - Object is not virtual disk storage. 1 - Object is virtual disk storage.

Table 8. QAEZDxxxxO (store object's attributes) (continued)

Field name	Field type	Field description
QEZCRTAUD	GRAPHIC (10)	<p>The auditing value associated with an object created in this directory. Valid values are:</p> <p>*NONE - No auditing occurs for this object when it is read or changed regardless of the user who is accessing the object.</p> <p>*USRPRF - Audit this object only if the current user is being audited. The current user is tested to determine if auditing should be done for this object. The user profile can specify if only change accesses are audited or if both read and change accesses are audited for this object.</p> <p>*CHANGE - Audit all change accesses to this object by all users on the system.</p> <p>*ALL - Audit all accesses to this object by all users on the system. All accesses are defined as a read or change operation.</p>
QEZCRTTIM	TIMESTAMP	The date and time at which the object was created.
QEZDIRIDX	INTEGER	The parent directory index.
QEZDIRTYP2	SMALLINT	<p>The format of the specified directory object. Valid values are:</p> <p>0 - The directory format is *TYPE1.</p> <p>1 - The directory format is *TYPE2.</p>
QEZDOM	GRAPHIC (10)	<p>The domain of the object. Valid values are:</p> <p>*SYSTEM - Object exists in system domain.</p> <p>*USER - Object exists in user domain.</p>
QEZDSTGOPT	SMALLINT	<p>This option should be used to determine how auxiliary storage is allocated by the system for the specified object. This option can only be specified for stream files in the "root" (/), QOpenSys and user-defined file systems. This option will be ignored for *TYPE1 byte stream files. Valid values are:</p> <p>0 - The auxiliary storage will be allocated normally. That is, as additional auxiliary storage is required, it will be allocated in logically sized extents to accommodate the current space requirement, and anticipated future requirements, while minimizing the number of disk I/O operations.</p> <p>1 - The auxiliary storage will be allocated to minimize the space used by the object. That is, as additional auxiliary storage is required, it will be allocated in small sized extents to accommodate the current space requirement. Accessing an object composed of many small extents may increase the number of disk I/O operations for that object.</p> <p>2 - The system will dynamically determine the optimum auxiliary storage allocation for the object, balancing space used versus disk I/O operations. For example, if a file has many small extents, yet is frequently being read and written, then future auxiliary storage allocations will be larger extents to minimize the number of disk I/O operations. Or, if a file is frequently truncated, then future auxiliary storage allocations will be small extents to minimize the space used. Additionally, information will be maintained on the stream file sizes for this system and its activity. This file size information will also be used to help determine the optimum auxiliary storage allocations for this object as it relates to the other objects sizes.</p>
QEZDTASIZE	BIGINT	The size in bytes of the data in this object. This size does not include object headers or the size of extended attributes associated with the object.

Table 8. QAEZDxxxxO (store object's attributes) (continued)

Field name	Field type	Field description
QEZEAS	BIGINT	Number of extended attributes associated with this object.
QEZEATATRS	BIGINT	Total number of bytes for all the extended attribute data.
QEZFILEID ¹	GRAPHIC (16)	The file ID for the object. An identifier associated with the object. A file ID can be used with Qp0lGetPathFromFileID() to retrieve an object's path name.
QEZFILEIDS	INTEGER	The 4 byte file ID of the file. This number uniquely identifies the object within a file system. This number cannot identify the object in the whole system.
QEZFILTYP2 ¹	SMALLINT	The format of the stream file (*STMF). Valid values are: 0 - The stream file format is *TYPE1. 1 - The stream file format is *TYPE2.
QEZFSID	BIGINT	The file system ID to which the object belongs. This number uniquely identifies the file system to which the object belongs.
QEZGENID	BIGINT	The generation ID associated with the file ID.
QEZGID	INTEGER	Group profiles are identified by a unique numeric group identification number (GID).
QEZINHSCN	GRAPHIC (1)	Whether the objects created in a directory will be scanned when exit programs are registered with any of the integrated file system scan-related exit points. Valid values are: x'00' - After an object is created in the directory, the object will not be scanned according to the rules described in the scan-related exit programs. Note: If the Scan file systems control (QSCANFCTL) value *NOPOSTRST is not specified when an object with this attribute is restored, the object will be scanned at least once after the restore. x'01' - After an object is created in the directory, the object will be scanned according to the rules described in the scan-related exit programs if the object has been modified or if the scanning software has been updated since the last time the object was scanned. x'02' - After an object is created in the directory, the object will be scanned according to the rules described in the scan-related exit programs only if the object has been modified since the last time the object was scanned. It will not be scanned if the scanning software has been updated. This attribute only takes effect if the Scan file systems control (QSCANFCTL) system value has *USEOCOATR specified. Otherwise, it will be treated as if the attribute is SCANNING_YES. Note: If the Scan file systems control (QSCANFCTL) value *NOPOSTRST is not specified when an object with this attribute is restored, the object will be scanned at least once after the restore.
QEZJAFTERI	SMALLINT	When journaling is active, the image of the object after a change is journaled. 0 - Object is not journaled with after images. 1 - Object is journaled with after images.

Table 8. QAEZDxxxxO (store object's attributes) (continued)

Field name	Field type	Field description
QEZJBEBORI	SMALLINT	When journaling is active, the image of the object before a change is journaled. 0 - Object is not journaled with before images. 1 - Object is journaled with before images.
QEZOPTENT	SMALLINT	When journaling is active, entries that are considered optional are journaled. The list of optional journal entries varies for each object type. 0 - Object is not journaled with optional entries. 1 - Object is journaled with optional entries.
QEZRJVASP	GRAPHIC(10)	The name of the ASP that contains the journal receiver needed to successfully apply journal changes. The valid values are: *SYSBAS - The journal receiver resides in the system or user ASP. ASP device - The name of the ASP device that contains the journal receiver.
QEZRJVLIB	GRAPHIC(10)	The name of the library that contains the journal receiver needed to successfully apply journal changes. This field is blank if the object has never been journaled.
QEZRJVNAM	GRAPHIC(10)	The oldest journal receiver needed to successfully apply journal changes. When the Apply information field is set to PARTIAL_TRANSACTION, the journal receiver contains the start of the partial transaction. Otherwise, the journal receiver contains the start of the save operation. This field is blank if the object has never been journaled.
QEZRJNID	GRAPHIC (10)	This field associates the object being journaled with an identifier that can be used on various journaling-related commands and APIs. This field is blank if the object has never been journaled.
QEZRJNLIB	GRAPHIC (10)	If the value of the journaling status is JOURNALED, then this field contains the name of the library containing the currently used journal. If the value of the journaling status is NOT_JOURNALED, then this field contains the name of the library containing the last used journal. This field is blank if the object has never been journaled.
QEZRJNNAM	GRAPHIC (10)	If the value of the journaling status is JOURNALED, then this field contains the name of the journal currently being used. If the value of the journaling status is NOT_JOURNALED, then this field contains the name of the journal last used for this object. This field is blank if the object has never been journaled.
QEZRJNSTR	TIMESTAMP	The number of seconds since the Epoch that corresponds to the last date and time for which the object had journaling started for it. This field has NULL as its value if the object has never been journaled.
QEZRJNSTS ¹	SMALLINT	Current journaling status of the object. This field will be one of the following values: 0 (NOT_JOURNALED) - The object is currently not being journaled. 1 (JOURNALED) - The object is currently being journaled.

Table 8. QAEZDxxxxO (store object's attributes) (continued)

Field name	Field type	Field description
QEZJSUBTRE	SMALLINT	<p>When this flag is returned, this object is a directory with integrated file system journaling subtree semantics.</p> <p>0 - The object is not journaled with subtree semantics.</p> <p>1 - The object is journaled with subtree semantics. New objects created within this directory's subtree will inherit the journaling attributes and options from this directory.</p>
QEZJTRNI	GRAPHIC(1)	<p>This field describes information about the current state of the object as it relates to commitment control boundaries. The valid values are:</p> <p>x'00' (NONE) - There are no partial transactions.</p> <p>x'01' (PARTIAL_TRANSACTION) - The object was restored with partial transactions. This object cannot be used until the Apply Journaled Changes (APYJRNCHG) or Remove Journaled Changes (RMVJRNCHG) command is used to complete or rollback the partial transactions.</p> <p>x'02' (ROLLBACK_ENDED) - The object had a rollback operation ended using the "End Rollback" option on the Work with Commitment Definition (WRKCMTDFN) screen. It is recommended that the object be restored as it cannot be used. As a last option, the Change Journaled Object (CHGJRNOBJ) command can be used to allow the object to be used. Doing this, however, can leave the object in an inconsistent state.</p>
QEZLANGID	GRAPHIC (3)	<p>A three character ID representing the language that the object name (field QEZOBJNAM) is in.</p>
QEZLOCAL	SMALLINT	<p>Whether an object is stored locally or stored on a remote system. The decision of whether an object is local or remote varies according to the respective file system rules. Objects in file systems that do not carry either a local or remote indicator are treated as remote. Valid values are:</p> <p>1 - The object's data is stored locally.</p> <p>2 - The object's data is on a remote system.</p>
QEZMLTSIG	SMALLINT	<p>Whether an object has more than one i5/OS digital signature. Valid values are:</p> <p>0 - The object has only one digital signature.</p> <p>1 - The object has more than one digital signature. If the QEZSYSSIG field has the value 1, at least one of the signatures is from a source trusted by the system.</p>
QEZMODE	INTEGER	<p>The file access mode and type. For more information about mode, see open() API.</p>

Table 8. QAEZDxxxxO (store object's attributes) (continued)

Field name	Field type	Field description
QEZMSTGOPT	SMALLINT	<p>This option should be used to determine how main storage is allocated and used by the system for the specified object. This option can only be specified for stream files in the "root" (/), QOpenSys and user-defined file systems. Valid values are:</p> <p>0 - The main storage will be allocated normally. That is, as much main storage as possible will be allocated and used. This minimizes the number of disk I/O operations since the information is cached in main storage.</p> <p>1 - The main storage will be allocated to minimize the space used by the object. That is, as little main storage as possible will be allocated and used. This minimizes main storage usage while increasing the number of disk I/O operations since less information is cached in main storage.</p> <p>2 - The system will dynamically determine the optimum main storage allocation for the object depending on other system activity and main storage contention. That is, when there is little main storage contention, as much storage as possible will be allocated and used to minimize the number of disk I/O operations. And when there is significant main storage contention, less main storage will be allocated and used to minimize the main storage contention. This option only has an effect when the storage pool's paging option is *CALC. When the storage pool's paging option is *FIXED, the behavior is the same as STG_NORMAL. When the object is accessed through a file server, this option has no effect. Instead, its behavior is the same as STG_NORMAL.</p>
QEZNLNK	INTEGER	The number of hard links to the object.
QEZNMCCSID	INTEGER	The CCSID in which the object name (field QEZOBJNAM) is represented.
QEZNONSAV	SMALLINT	<p>Whether the object can be saved or not. Valid values are:</p> <p>0 - Object will be saved.</p> <p>1 - Object will not be saved. Additionally, if this object is a directory, none of the objects in the directory's subtree will be saved unless they were explicitly specified as an object to be saved. The subtree includes all subdirectories and the objects within those subdirectories.</p>
QEZOBJLEN	INTEGER	Number of bytes contained in the object name (field QEZOBJNAM).
QEZOBJNAM ¹	VARGRAPHIC (1024)	The object name. ²
QEZOBJTYPE ¹	GRAPHIC (10)	The object type.
QEZOFLOW	SMALLINT	<p>Indicates if the object has overflowed the auxiliary storage pool it resides in. Valid values are:</p> <p>0 - Auxiliary storage pool is not overflowed.</p> <p>1 - Auxiliary storage pool is overflowed.</p>
QEZOWN ¹	GRAPHIC (10)	<p>The name of the user profile that is the owner of the object or the following special value:</p> <p>*NOUSRPRF - This special value is used by the Network File System to indicate that there is no user profile on the local System i platform with a user ID (UID) matching the UID of the remote object.</p>

Table 8. QAEZDxxxxO (store object's attributes) (continued)

Field name	Field type	Field description
QEZOWNPGP	GRAPHIC (10)	The name of the user profile that is the primary group of the object or the following special values: *NONE - The object does not have a primary group *NOUSRPRF - This special value is used by the Network File System to indicate that there is no user profile on the local system with a group ID (GID) matching the GID of the remote object.
QEZPCARC	SMALLINT	Whether the object has changed since the last time the object was examined. 0 - The object has not changed. 1 - The object has changed.
QEZPCHID ¹	SMALLINT	Whether the object can be displayed using an ordinary directory listing. 0 - The object is not hidden. 1 - The object is hidden.
QEZPCREAD	SMALLINT	Whether the object can be written to or deleted, have its extended attributes changed or deleted, or have its size changed. Valid values are: 0 - The object can be changed. 1 - The object cannot be changed.
QEZPCSYS	SMALLINT	Whether the object is a system file and is excluded from normal directory searches. 0 - The object is not a system file. 1 - The object is a system file.
QEZPRMLNK	SMALLINT	When an object has several names, this field will be set only for the first name found.
QEZRDEV	BIGINT	If the object represents a device special file, the real device it represents.
QEZREGION	GRAPHIC (2)	A two-character ID representing the country of the object name (field QEZOBJNAM). This ID affects actions that tend to be defined by the location of the action, such as collating sequence.
QEZSBINARY	GRAPHIC (1)	This indicates if the object has been scanned in binary mode when it was previously scanned. This field will be one of the following values: x'00' - The object was not scanned in binary mode. x'01' - The object was scanned in binary mode. If the object scan status is SCAN_SUCCESS, then the object was successfully scanned in binary. If the object scan status is SCAN_FAILURE, then the object failed the scan in binary.
QEZSCCSID1	INTEGER	This indicates if the object has been scanned in the listed CCSID when it was previously scanned. If the object scan status is SCAN_SUCCESS, then the object was successfully scanned in this CCSID. If the object scan status is SCAN_FAILURE, then the object failed the scan in this CCSID. A value of 0 means this field does not apply.
QEZSCCSID2	INTEGER	This indicates if the object has been scanned in the listed CCSID when it was previously scanned. If the object scan status is SCAN_SUCCESS, then the object was successfully scanned in this CCSID. If the object scan status is SCAN_FAILURE, then this field will be 0. A value of 0 means this field does not apply.

Table 8. QAEZDxxxxO (store object's attributes) (continued)

Field name	Field type	Field description
QEZSCN	GRAPHIC (1)	<p>Whether the object will be scanned when exit programs are registered with any of the integrated file system scan-related exit points.</p> <p>Valid values are:</p> <p>x'00' (SCANNING_NO) - The object will not be scanned according to the rules described in the scan-related exit programs. Note: If the Scan file systems control (QSCANFCTL) value *NOPOSTRST is not specified when an object with this attribute is restored, the object will be scanned at least once after the restore.</p> <p>x'01' (SCANNING_YES) - The object will be scanned according to the rules described in the scan-related exit programs if the object has been modified or if the scanning software has been updated since the last time the object was scanned.</p> <p>x'02' (SCANNING_CHGONLY) - The object will be scanned according to the rules described in the scan-related exit programs only if the object has been modified since the last time the object was scanned. It will not be scanned if the scanning software has been updated. This attribute only takes effect if the Scan file systems control (QSCANFCTL) system value has *USEOCOATR specified. Otherwise, it will be treated as if the attribute is SCANNING_YES. Note: If the Scan file systems control (QSCANFCTL) value *NOPOSTRST is not specified when an object with this attribute is restored, the object will be scanned at least once after the restore.</p>
QEZSIG ¹	SMALLINT	<p>Whether an object has an i5/OS digital signature. Valid values are:</p> <p>0 - The object does not have an i5/OS digital signature.</p> <p>1 - The object does have an i5/OS digital signature.</p>
QEZSSIGDF	GRAPHIC (1)	<p>The scan signatures give an indication of the level of the scanning software support.</p> <p>When an object is in an independent ASP group, the object scan signature is compared to the associated independent ASP group scan signature. When an object is not in an independent ASP group, the object scan signature is compared to the global scan signature value. This field will be one of the following values:</p> <p>x'00' - The compared signatures are not different.</p> <p>x'01' - The compared signatures are different.</p>

Table 8. QAEZDxxxxO (store object's attributes) (continued)

Field name	Field type	Field description
QEZSSTATUS	GRAPHIC (1)	<p>The scan status associated with this object. This field will be one of the following values:</p> <p>x'00' (SCAN_REQUIRED) - A scan is required for the object either because it has not yet been scanned by the scan-related exit programs, or because the objects data or CCSID has been modified since it was last scanned. Examples of object data or CCSID modifications are: writing to the object, directly or through memory mapping, truncating the object, clearing the object, and changing the objects CCSID attribute.</p> <p>x'01' (SCAN_SUCCESS) - The object has been scanned by a scan-related exit program, and at the time of that last scan request, the object did not fail the scan.</p> <p>x'02' (SCAN_FAILURE) - The object has been scanned by a scan-related exit program, and at the time of that last scan request, the object failed the scan and the operation did not complete. Once an object has been marked as a failure, it will not be scanned again until the object's scan signature is different than the global scan key signature or independent ASP group scan key signature as appropriate. Therefore, subsequent requests to work with the object will fail with a scan failure indication. Examples of requests which will fail are opening the object, changing the CCSID of the object, copying the object.</p> <p>x'05' (SCAN_PENDING_CVN) - The object is not in a *TYPE2 directory, and therefore will not be scanned until the directory is converted.</p> <p>x'06' (SCAN_NOT_REQUIRED) - The object does not require any scanning because the object is marked to not be scanned.</p>
QEZSTGFREE ¹	SMALLINT	<p>Whether the object's data has been moved offline, freeing its online storage. Valid values are:</p> <p>0 - The object's data is not offline.</p> <p>1 - The object's data is offline.</p>
QEZSYSARC	SMALLINT	<p>Whether the object has changed and needs to be saved. It is set on when an object's change time is updated, and set off when the object has been saved.</p> <p>0 - The object has not changed and does not need to be saved.</p> <p>1 - The object has changed and does need to be saved.</p>
QEZSYSSIG	SMALLINT	<p>Whether the object was signed by a source that is trusted by the system. Valid values are:</p> <p>0 - None of the signatures came from a source that is trusted by the system.</p> <p>1 - The object was signed by a source that is trusted by the system. If the object has multiple signatures, at least one of the signatures came from a source that is trusted by the system.</p>
QEZUDATE	TIMESTAMP	<p>The number of seconds since the Epoch that corresponds to the date the object was last used. This field is zero when the object is created. If usage data is not maintained for the i5/OS type or the file system to which an object belongs, this field is zero.</p>

Table 8. QAEZDxxxxO (store object's attributes) (continued)

Field name	Field type	Field description
QEZUDCOUNT	INTEGER	The number of days an object has been used. Usage has different meanings according to the specific file system and according to the individual object types supported within a file system. Usage can indicate the opening or closing of a file or can refer to adding links, renaming, restoring, or checking out an object. This count is increased each day that an object is used and is reset to zero by calling the Qp0lSetAttr() API.
QEZUDFTYP2	SMALLINT	The default file format of stream files (*STMF) created in the user-defined file system. Valid values are: 0 - The stream file format is *TYPE1. 1 - The stream file format is *TYPE2.
QEZUID	INTEGER	Each user on the system must have a unique numeric user identification number (UID).
QEZURESET	INTEGER	The number of seconds since the Epoch that corresponds to the date the days used count was last reset to zero (0). This date is set to the current date when the Qp0lSetAttr() API is called to reset the Days used count to zero.
Notes:		
<ol style="list-style-type: none"> 1. This field is included in the subset of fields used by the PRTDIRINF command. 2. In this field, only the object name is stored. The rest of the path name is stored in field QEZDIRNAM1 if the length of the directory name is below 1 KB (equals 1024 bytes) or in QEZDIRNAM2 if the directory names are above 1 KB (equals 1024 bytes). 		

The following table is an example of a table that lists the directories processed by the RTVDIRINF command.

Table 9. QAEZDxxxD (store directory's attributes)

Field name	Field type	Field description
QEZDFID	INTEGER	The file ID of the directory.
QEZDIRFID	GRAPHIC (16)	The file ID of the directory. An identifier associated with the object. A file ID can be used with Qp0lGetPathFromFileID() to retrieve an object's path name.
QEZDIRFSID	BIGINT	The file system ID of the directory.
QEZDIRGID	BIGINT	The generation ID.
QEZDIRIDX	INTEGER	Identifier of path name (applies only to directories).
QEZDIRLEN ¹	INTEGER	Length of directory's path name.
QEZDIRNAM1 ¹	VARGRAPHIC (1024)	The parent directory path. Only used when path length is below 1 KB (1024 bytes).
QEZDIRNAM2 ¹	DBCLOB (16M)	The parent directory path. Only used when path length is above 1 KB (1024 bytes) long. Can store paths up to 16 MB long.
QEZDRCCSID	INTEGER	The directory CCSID.
QEZDREGION	GRAPHIC (2)	The directory path region ID.
QEZLANGID	GRAPHIC (3)	The directory path language ID.
Note:		
<ul style="list-style-type: none"> • This field is included in the subset of fields used by the PRTDIRINF command. 		

The following table shows the information that the RTVDIRINF command stores regarding the files it has created when it runs. If the file that contains this information does not exist, the RTVDIRINF command creates it; when the command runs on subsequent occasions, the information is appended to the existing file. The PRTDIRINF command uses this information to determine which database files were used to store information that was retrieved by different instances of the RTVDIRINF command.

Table 10. QUSRSYS/QAEZDBFILE (store files created)

Field name	Field type	Field description
QEZDIRFILE ¹	VARGRAPHIC (20)	The name of the file generated to store the directory's indexes.
QEZDIRSRC	VARGRAPHIC (5000)	Path specified in DIR parameter (RTVDIRINF).
QEZENDTIME	TIMESTAMP	Date/Time when RTVDIRINF was completed.
QEZLIB ¹	VARGRAPHIC (20)	Library where both generated files reside.
QEZOBJFILE ¹	VARGRAPHIC (20)	The name of the file generated to store the object's attributes.
QEZPLANGID	GRAPHIC (3)	The path language ID
QEZPRCCSID	INTEGER	The path CCSID.
QEZPREGION	GRAPHIC (2)	The path region ID.
QEZSTRTIME	TIMESTAMP	Date/Time when RTVDIRINF was submitted.
Note:		
<ul style="list-style-type: none"> This field is included in the subset of fields used by the PRTDIRINF command. 		

Related information

Retrieve Directory Information (RTVDIRINF) command

Qp0lGetPathFromFileID()--Get Path Name of Object from Its File ID API

Qp0lSetAttr()--Set Attributes API

Apply Journaled Changed (APYJRNCHG) command

Remove Journaled Changes (RMVJRNCHG) command

Change Journaled Object (CHGJRNOBJ) command

Print Directory Information (PRTDIRINF) command

Accessing the data of RTVDIRINF:

There exist several options for accessing the data in the tables.

Listed below are ways that you can access the data created by the Retrieve Directory Information (RTVDIRINF) command:

- Using the Print Directory Information (PRTDIRINF) command
This command is used to print directory information about objects and directory information in the integrated file system. The information it will print is already stored in the database file specified by the user in the RTVDIRINF command.
- Using any program or command provided by IBM that can run queries over a DB2[®] table on the i5/OS operating system.
Some of the more common tools are the Start SQL Interactive Session (STRSQL) command and the iSeries Navigator.
For example, if you want to select objects in a specific path (previously collected by the RTVDIRINF command) that have an allocation size larger than 10 KB, you can run a query like this:

```
SELECT QEZOBJNAM, QEZALCSIZE FROM library_name/QAEZDxxxx0 WHERE QEZALCSIZE > 10240
```
- You can make your own programs and access the database tables by using any valid DB methods.

Related information

Print Directory Information (PRTDIRINF) command

Start SQL Interactive Session (STRSQL) command

Embedded SQL programming

SQL call level interface

Using the data of RTVDIRINF:

Here are examples that display why the data is important or how you can use the data that is produced from each of the three tables.

- For Table 8 on page 70, you can make queries to create reports or statistics based on any of the fields within this table. PRTDIRINF does not include reports based on all of the fields. Instead, a subset will be used.
- The data from Table 9 on page 80 contains all the directories within the path specified in DIR parameter of the RTVDIRINF command. If you want to know specific attributes about the path name, for example the CCSID, language ID, or length, then this data is useful. Also, each directory stored in this table has a unique value, or index that identifies it. In Table 8 on page 70, you can find the same field, QEZDIRIDX, that will tell you which objects belong to which directory. To find which objects belong to which directory, you can make a query using joins. For example, the following query statement selects the names of all objects existing in directory "/MYDIR":

```
SELECT QEZOBJNAM FROM library_name/QAEZxxxx0, library_name/QAEZxxxxD WHERE QEZDIRNAM1 = "/MYDIR" AND library_name/QAEZxxxx0.QEZDIRIDX=library_name/QAEZxxxxD.QEZDIRIDX
```

- Table 10 on page 81 is mostly used by the PRTDIRINF command to obtain specific data about RTVDIRINF runs. Examples of this are: The names of the tables created, the library where the tables reside, and the starting and ending time of processing. You might use this table to know when a RTVDIRINF was issued or what tables must be searched in order to query them.

Accessing using APIs

You can use application programming interfaces (APIs) to access the integrated file system.

Related reference

“Performing operations using APIs” on page 104

Many of the application programming interfaces (APIs) that perform operations on integrated file system objects are in the form of C language functions.

Accessing using iSeries Navigator

iSeries Navigator is the graphical user interface for managing and administering your systems from your Windows desktop. iSeries Navigator makes operation and administration of your system easier and more productive.

For instance, you can copy a user profile onto another system by dragging the user profile from one system to another. Wizards guide you through setting up security and TCP/IP services and applications.

There are many tasks you can perform using iSeries Navigator. Listed below are some common file system tasks to help you get started:

Working with files and folders

- “Creating a folder” on page 120
- “Removing a folder” on page 120
- “Checking in a file” on page 120
- “Checking out a file” on page 120
- “Setting permissions” on page 122

- “Setting up file text conversion” on page 122
- “Sending a file or folder to another system” on page 123
- “Changing options for a package definition” on page 123
- “Scheduling a date and time to send your file or folder” on page 124
- “Setting whether objects should be scanned or not” on page 125

Working with file shares

- “Creating a file share” on page 124
- “Changing a file share” on page 124

Working with user-defined file systems

- “Creating a new user-defined file system” on page 124
- “Mounting a user-defined file system” on page 124
- “Unmounting a user-defined file system” on page 125

Journaling objects

- “Starting journaling” on page 95
- “Ending journaling” on page 95

Accessing using iSeries NetServer

iSeries Support for Windows Network Neighborhood (iSeries NetServer) is a function that enables Windows clients to access i5/OS shared directory paths and shared output queues. iSeries NetServer allows PCs that run Windows software to seamlessly access data and printers that are managed by your System i platform.

About this task

PC clients on a network use the file and print sharing functions that are included in their operating systems. This means that you do not need to install any additional software on your PC to use iSeries NetServer.

Linux clients with the Samba client software installed can also seamlessly access data and printers through iSeries NetServer. Shared iSeries NetServer directories can be mounted on Linux clients as Samba file systems (smbfs) in a similar manner to mounting NFS file systems that have been exported from iSeries.

An iSeries NetServer file share is a directory path that iSeries NetServer shares with clients on the System i network. A file share can consist of any integrated file system directory on the system. Before you can work with file sharing using iSeries NetServer, you must create an iSeries NetServer file share, and, if necessary, change an iSeries NetServer file share using iSeries Navigator.

To access integrated file system file shares using iSeries NetServer:

1. Right-click **Start**, and select **Explore** to open Windows Explorer on your Windows PC.
2. Open the Tools menu, and select **Map network drive**.
3. Select a letter of a free drive for the file share (such as the I:\ drive).
4. Enter the name of an iSeries NetServer file share. For example, you can enter the following syntax:
\\QSYSTEM1\Sharename

Note: QSYSTEM1 is the system name of iSeries NetServer, and Sharename is the name of the file share you want to use.

5. Click **OK**.

Results

Note: When connecting using iSeries NetServer, the system name might be different from the name used by the iSeries Access Family. For example the iSeries NetServer name might be QAS400X, and the path to work with files might be \\QAS400X\QDLS\MYFOLDER.FLR\MYFILE.DOC. However, the iSeries Access Family name might be AS400X, and the path to work with files might be \\AS400X\QDLS\MYFOLDER.FLR\MYFILE.DOC.

You choose which directories to share with the network using iSeries NetServer. Those directories appear as the first level under the system name. For example, if you share the /home/fred directory with the name fredsdir, a user can access that directory from the PC with the name \\QAS400X\FRESDIR, or from a Linux client with the name //qas400x/fredsdir.

The "root" (/) file system provides much better performance for PC file serving than other i5/OS file systems. You might want to move files to the "root" (/) file system. See "Moving files or folders to another file system" on page 121 for more information

Related information

iSeries NetServer

iSeries NetServer file shares

Accessing iSeries NetServer file shares with a Windows PC client

Accessing using File Transfer Protocol

The File Transfer Protocol (FTP) client allows you to transfer files that are found on your System i platform, including those in the "root" (/), QOpenSys, QSYS.LIB, independent ASP QSYS.LIB, QOPT, and QFileSvr.400 file systems.

You can also transfer folders and documents in the document library services (QDLS) file system. The FTP client can be run interactively in an unattended batch mode where client subcommands are read from a file and the responses to these subcommands are written to a file. It also includes other features for manipulating files on your system.

You can use FTP support to transfer files to and from any of the following file systems:

- "root" (/) file system
- Open systems file system (QOpenSys)
- Library file system (QSYS.LIB)
- Independent ASP QSYS.LIB file system
- Document library services file system (QDLS)
- Optical file system (QOPT)
- Network File System (NFS)
- NetWare file system (QNetWare)
- iSeries NetClient file system (QNTC)

However, be aware of the following restrictions:

- The integrated file system limits FTP support to transferring file data only. You cannot use FTP to transfer attribute data.
- QSYS.LIB and independent ASP QSYS.LIB file systems limit FTP support to physical file members, source physical file members, and save files. You cannot use FTP to transfer other object types, such as programs (*PGM). However, you can save other object types to a save file, transfer the save file, and then restore the objects.

Related information

File Transfer Protocol

Accessing using a PC

If your PC is connected to a System i product, you can interact with the directories and objects of the integrated file system as if they were stored on your PC.

You can copy objects between directories by using the drag-and-drop capability of Windows Explorer. As needed, you can actually copy an object from your system to the PC by selecting the object in the system drive and dragging the object to the PC drive.

Any objects that are copied between a System i product and PCs by using the Windows interface can be automatically converted between EBCDIC (extended binary-coded decimal interchange code) and ASCII (American National Standard Code for Information Interchange). The iSeries Access Family can be configured to automatically perform this conversion, and can even specify that the conversion be performed on files with a specific extension.

Depending on the type of object, you can use PC interfaces and PC applications to work with it. For example, a stream file containing text can be edited using a PC editor.

If you are connected to a System i product using your PC, the integrated file system makes your system's directories and objects available to the PC. PCs can work with files in the integrated file system by using file sharing clients built into the Windows operating systems, an FTP client, or iSeries Navigator (a part of iSeries Access Family). Your PC uses Windows file sharing clients to access iSeries NetServer, which runs on your system.

Related concepts

"Accessing using iSeries Navigator" on page 82

iSeries Navigator is the graphical user interface for managing and administering your systems from your Windows desktop. iSeries Navigator makes operation and administration of your system easier and more productive.

Related tasks

"Accessing using iSeries NetServer" on page 83

iSeries Support for Windows Network Neighborhood (iSeries NetServer) is a function that enables Windows clients to access i5/OS shared directory paths and shared output queues. iSeries NetServer allows PCs that run Windows software to seamlessly access data and printers that are managed by your System i platform.

Related reference

"Accessing using File Transfer Protocol" on page 84

The File Transfer Protocol (FTP) client allows you to transfer files that are found on your System i platform, including those in the "root" (/), QOpenSys, QSYS.LIB, independent ASP QSYS.LIB, QOPT, and QFileSvr.400 file systems.

Converting directories from *TYPE1 to *TYPE2

The "root" (/), QOpenSys, and user-defined file systems (UDFS) in the integrated file system support the *TYPE2 directory format as of OS/400 V5R1.

The *TYPE2 directory format is an enhancement of the original *TYPE1 directory format. *TYPE2 directories have a different internal structure from *TYPE1 directories and provide improved performance and reliability.

Shortly after i5/OS V5R3M0 or a later release is installed, the conversion to *TYPE2 directories automatically begins for any of the file systems that have not yet been converted to support *TYPE2 directories. This conversion should not significantly impact your system activity.

Overview of *TYPE1 to *TYPE2 conversion

The "root" (/), QOpenSys, and user-defined file systems (UDFS) in the integrated file system support the *TYPE2 directory format as of OS/400 V5R1.

The *TYPE2 directory format is an enhancement of the original *TYPE1 directory format. *TYPE2 directories have a different internal structure from *TYPE1 directories and provide improved performance and reliability. In addition to improved performance and reliability, new functions such as integrated file system scanning support are only available for objects in *TYPE2 directories. See "Scanning support" on page 19 for more information.

Shortly after i5/OS V5R3M0 operating system or a later release is installed, the conversion to *TYPE2 directories will automatically begin for any of the file systems which have not yet been converted to support *TYPE2 directories. This conversion should not significantly impact your system activity as it will run in a low priority background job.

If the conversion function has not yet completed, and the system has a normal or abnormal IPL, the conversion function will resume when the IPL is completed. The conversion will restart on every initial program load (IPL) until all eligible file systems have been fully converted.

The file systems that are eligible for this automatic conversion are "root" (/), QOpenSys and the user-defined file systems for ASPs 1 through 32.

Note: You can avoid the automatic conversion to *TYPE2 directories if you convert the file systems before installing V5R3 operating system or a later release.

Related concepts

"*TYPE2 directories" on page 10

The "root" (/), QOpenSys, and user-defined file systems (UDFS) in the integrated file system support the *TYPE2 directory format. The *TYPE2 directory format is an enhancement of the original *TYPE1 directory format.

Related reference

"Conversion status determination"

Shortly after i5/OS V5R3M0 operating system or a later release is installed, the conversion to *TYPE2 directories will automatically begin for any of the file systems which have not yet been converted to support *TYPE2 directories. This conversion processing will take place in a secondary thread of the QFILESYS1 system job.

"Tips: Independent ASP" on page 89

If the user-defined file systems in an independent ASP have not yet been converted to the *TYPE2 directory format, they will be converted the first time the independent ASP is varied on to a system installed with V5R2 or later of the operating system.

Conversion considerations

Here are several things to consider during the conversion process.

Conversion status determination

Shortly after i5/OS V5R3M0 operating system or a later release is installed, the conversion to *TYPE2 directories will automatically begin for any of the file systems which have not yet been converted to support *TYPE2 directories. This conversion processing will take place in a secondary thread of the QFILESYS1 system job.

To determine the status of the conversion processing, you can use the Convert Directory (CVTDIR) command as follows:

```
CVTDIR OPTION(*CHECK)
```

This invocation of the CVTDIR command lists the current directory format for the "root" (/), QOpenSys and UDFS file systems and if the file system is currently being converted. Additionally, it lists the current priority of the conversion function, the file system currently being converted by the system, the number of links that have been processed for that file system, and the percentage of directories which have been processed for that file system. The system starts the conversion function with a very low priority (99) so that the conversion function does not significantly impact system activity. However, you can change the priority of the conversion function by using the *CHGPTY value for the OPTION parameter of the CVTDIR command. See CVTDIR for additional information about this parameter specification.

Because the QFILESYS1 job is processing the conversion, you can display the QFILESYS1 job log for messages indicating any problems with the conversion. Additionally, various progress messages are sent about the file system conversions. These messages include information such as: the file system being converted, the number of links which have been processed in the file system, the percentage of directories that have been processed in the file system, and so on. All of the error and many of the progress messages are also sent to the QSYSOPR message queue. Therefore, for future reference, it is good practice to ensure that the QHST logs or QFILESYS1 job logs are preserved, which contain these messages. After the file systems have been fully converted, and the integrated file system is working as expected, you can delete this historical information.

Related information

Convert Directory (CVTDIR) command

User profiles creation

The conversion function creates a user profile that is used while the conversion function is running. This user profile has the name QP0FCWA. It is used by the conversion function to own converted directories in the file system if the original owner is unable to own their directories.

The user profile is deleted when the conversion has completed, if possible. Message CPIA08B is sent to the QFILESYS1 job log and the QSYSOPR message queue if ownership of a directory is given to this user profile.

Related reference

"Changing the owner of a directory" on page 88

If the user profile that owns the *TYPE1 directory is unable to own the *TYPE2 directory that is created, the owner of the *TYPE2 directory is set to the alternate user profile.

Objects renamed

*TYPE2 directories require the link names to be valid UTF-16 names.

The naming rule of *TYPE2 directories differs from *TYPE1 directories, which have UCS2 Level 1 names. For this reason, invalid or duplicate names might be found during a directory conversion. When a name is found to be invalid or a duplicate, the name is changed to a unique, valid, UTF-16 name, and message CPIA08A is sent to the QFILESYS1 job log and the QSYSOPR message queue listing the original name and the new name. Combined characters or invalid surrogate character pairs contained in a name might cause an object to be renamed.

For more information about UTF-16, refer to the Unicode home page (www.unicode.org ).

Combined characters:

Some characters can be made up of more than one Unicode character.

For example, characters that have an accent (for example, é or à) or an umlaut (for example, ä or ö) need to be changed, or *normalized*, to a common format before they are stored in the directory, so that all objects have a unique name. Normalizing a combined character is a process by which the character is put in a known and predictable format. The format chosen for *TYPE2 directories is the *canonical composed form*. If there are two objects in a *TYPE1 directory that contain the same combined characters, they are

normalized to the same name. This causes a collision, even if one object contains composed combined characters and the other object contains decomposed combined characters. Therefore, one of them has its name changed before it is linked in the *TYPE2 directory.

Surrogate characters:

Some characters do not have a valid representation in Unicode.

These characters have some special values; they are made up of two Unicode characters in two specific ranges such that the first Unicode character is in one range (for example 0xD800-0xD8FF) and the second Unicode character is in the second range (for example 0xDC00-0xDCFF). This is called a surrogate pair.

If one of the Unicode characters are missing or if they are out of order, (only a partial character), it is an invalid name. Names of this type have been allowed in *TYPE1 directories, but are not allowed in *TYPE2 directories. In order for the conversion function to continue, the name is changed before the object is linked into the *TYPE2 directory if a name containing one of these invalid names is found.

User profile considerations

While the conversion is running, every attempt is made to ensure that the same user profile that owns any *TYPE1 directories continues to own the corresponding *TYPE2 directories.

Since the *TYPE1 and *TYPE2 directories momentarily exist at the same time, this impacts the amount of storage owned by the user profile and the number of entries in the user profile.

Changing maximum storage for a user profile:

During the directory conversion processing, a number of directories that momentarily exist in both formats at the same time are owned by the same user profile.

If during conversion processing the maximum storage limit for the user profile is reached, the limit for the user profile is increased. Message CPIA08C is sent to the QFILESYS1 joblog and the QSYSOPR message queue.

Changing the owner of a directory:

If the user profile that owns the *TYPE1 directory is unable to own the *TYPE2 directory that is created, the owner of the *TYPE2 directory is set to the alternate user profile.

Message CPIA08B is sent to the QFILESYS1 job log and the QSYSOPR message queue, and conversion continues.

If the user profile that owns the *TYPE1 directory is unable to own the *TYPE2 directory that is created, the owner of the *TYPE2 directory is set to the alternate user profile. Message CPIA08B is sent to the QFILESYS1 job log and the QSYSOPR message queue, and conversion continues.

Related reference

“User profiles creation” on page 87

The conversion function creates a user profile that is used while the conversion function is running. This user profile has the name QP0FCWA. It is used by the conversion function to own converted directories in the file system if the original owner is unable to own their directories.

Auxiliary storage requirements

Auxiliary storage requirements should be considered while the system is converting the directories in a file system to the *TYPE2 format.

Here are several considerations regarding auxiliary storage requirements:

- The final size of the directories after they have been converted to the *TYPE2 format

- Additional storage required while the conversion function is running

In many cases, the final size of a *TYPE2 directory is smaller than a *TYPE1 directory. Typically, *TYPE2 directories that have less than 350 objects require less auxiliary storage than *TYPE1 directories with the same number of objects. *TYPE2 directories with more than 350 objects are ten percent larger (on average) than *TYPE1 directories.

While the conversion function is running, additional storage is required. The conversion function requires that the directories have both a *TYPE1 version and a *TYPE2 version in existence simultaneously.

Note: Before installing the i5/OS V5R3M0 operating system or a later release, you might want to consider running the *ESTIMATE option on the OS/400 V5R2 (CVTDIR) command, because it can provide a conservative estimate of the amount of auxiliary storage needed during conversion.

Related information

Convert Directory (CVTDIR) command

Tips: Symbolic link

Symbolic links are objects within the integrated file system that contain a path to another object.

There are some instances during conversion when the name of an object can be changed. If one of the elements of the path within a symbolic link is renamed during conversion, then the contents of the symbolic link no longer point to the object.

Related concepts

“Link” on page 11

A *link* is a named connection between a directory and an object. A user or a program can tell the system where to find an object by specifying the name of a link to the object. A link can be used as a path name or as part of a path name.

Related reference

“Objects renamed” on page 87

*TYPE2 directories require the link names to be valid UTF-16 names.

Related information

symlink()--Make Symbolic Link

Tips: Independent ASP

If the user-defined file systems in an independent ASP have not yet been converted to the *TYPE2 directory format, they will be converted the first time the independent ASP is varied on to a system installed with V5R2 or later of the operating system.

For planning purposes, an estimate function is provided in OS/400 V5R1 to provide information about the length of time that a conversion will run. Before varying on the independent ASP to the operating system running V5R2 or later, run the following API on your V5R1 operating system when the independent ASP (named ASP_NAME) is varied on and active:

```
CALL QP0FCVT2 (*ESTIMATE ASP_NAME *TYPE2)
```

Tips: Saving and restoring

Directories that exist as *TYPE1 can be saved and restored in a file system that has been converted to *TYPE2.

Likewise, directories that exist as *TYPE2 can be saved and restored in a file system that is *TYPE1 format, provided none of the *TYPE1 limits have been exceeded when the directory existed as a *TYPE2 directory.

l **Tips: Reclaiming integrated file system objects**

l While the system is converting the "root" (/), QOpenSys and user ASP UDFS file systems to support the *TYPE2 directory format, the Reclaim Storage (RCLSTG) and Reclaim Object Links (RCLLNK) commands cannot be run on any integrated file system directories including those in independent ASPs.

l The OMIT(*DIR) parameter value can be used on the RCLSTG command to omit integrated file system directories and allow the objects that are not related to integrated file system to be reclaimed.

l **Related concepts**

l "Reclaim operation of the "root" (/), QOpenSys, and user-defined file systems" on page 96
l Reclaiming the "root" (/), QOpenSys, and user-defined file systems can be accomplished using the Reclaim Object Links (RCLLNK) and Reclaim Storage (RCLSTG) commands.

l **Related information**

l Reclaim Storage (RCLSTG) command
l Reclaim Object Links (RCLLNK) command

Integrated file system scanning

Objects in the "root" (/), QOpenSys and user ASP UDFS file systems will not be scanned using the integrated file system scan-related exit points until the file systems have fully converted to the *TYPE2 directory format.

The scan-related attributes can be set for objects in *TYPE1 and *TYPE2 directories to specify whether the objects are to be scanned or not, even if the file system is not fully converted.

While the system converts objects from the *TYPE1 directory format to the *TYPE2 directory format, the Scan control system value Scan on next access after object has been restored is taken into consideration as if the converted object was being restored. For example, if the Scan on next access after object has been restored value is specified while the conversion is in progress, then an object that was in a *TYPE1 directory, and had the the object will not be scanned attribute specified, will be scanned at least once after the file system has been fully converted.

Related concepts

"Scanning support" on page 19
With the i5/OS operating system, you can scan integrated file system objects.

"Related system values" on page 20

Related to this scan support are two system values. You can use these two system values to establish the scanning environment you want for your system.

Journaling objects

The primary purpose of journaling is to enable you to recover the changes to an object that have occurred since the object was last saved. Additionally, a key use of journaling is to assist in the replication of object changes to another system either for high availability or workload balancing.

This information will provide a brief overview of journal management, as well as provide considerations for journaling integrated file system objects and a description of journaling support for integrated file system objects.

Related information

Journal management

Journaling overview

These topics introduce journaling support for integrated file system objects.

Related information

Journal management

Journal management

The main purpose for journal management is to enable you to recover the changes to an object that have occurred since the object was last saved.

You can also use journal management for:

- An audit trail of activity that occurs for objects on the system
- Recording activity that has occurred for objects other than those you can journal
- Quicker recovery when restoring from save-while-active media
- Assist in the replication of object changes to another system either for high availability or workload balancing
- Assistance in testing application programs

You can use a journal to define what objects you want to protect with journal management. In the integrated file system, you can journal stream files, directories, and symbolic links. Only objects in "root" (/), QOpenSys, and UDFS file systems are supported.

Related concepts

"Objects you should journal"

There are certain questions to consider when deciding whether to journal an integrated file system object.

Objects you should journal

There are certain questions to consider when deciding whether to journal an integrated file system object.

You need to consider the following questions to determine which objects you need to journal:

- How much does the object change? An object with a high volume of changes between save operations is a good candidate for journaling.
- How difficult is it to reconstruct the changes made to an object? Are many changes made to the object without written records? For example, an object used for telephone order entries is more difficult to reconstruct than an object used for orders that arrive in the mail on order forms.
- How critical is the information in the object? If the object had to be restored back to the last save operation, what effect does the delay in reconstructing changes have on the business?
- How does the object relate to other objects on the system? Although the data in a particular object might not change often, that object's data might be critical to other more dynamic objects on the system. For example, many objects depend on a customer master file. If you are reconstructing orders, the customer master file must include new customers or changes in credit limits that have taken place since the previous save operation.

Journalled integrated file system objects

Some integrated file system object types can be journaled using i5/OS journaling support.

The object types supported are stream files, directories, and symbolic links. The "root" (/), QOpenSys, and UDFS are the only file systems that support journaling of these object types. Integrated file system objects can be journaled using either the traditional system interface (CL commands or APIs) or by using iSeries Navigator. You can start journaling and end journaling through iSeries Navigator, as well as display journaling status information.

Note: Memory-mapped stream files, virtual volume files, and stream files that are used as Integrated xSeries Server for iSeries (IXS) network storage spaces cannot be journaled. Directories that can contain block special file objects cannot be journaled. Examples of these are: /dev/QASP01, /dev/QASP22, and /dev/IASPNAME.

The following list summarizes journaling support in the integrated file system:

- You can use both generic commands and APIs to perform journal operations on the supported object types. These interfaces generally accept object identification in the form of a path name, file ID, or both.
- Some journal operation commands, including Start Journaling, End Journaling, Change Journaling and Apply Journalized Changes, may be performed on entire subtrees of integrated file system objects. You can optionally use the include and exclude lists that may use wildcard patterns for object names. For example, you can use the Start Journaling command to specify to start on all objects in the tree `"/MyCompany"` that match the pattern `"*.data"` but that excludes any objects matching the patterns `"A*.data"` and `"B*.data"`.
- Journaling support on directories includes directory operations such as adding links, removing links, creating objects, renaming objects, and moving objects within the directory.

Journalized directories support an attribute that can be set to cause new objects in the subtree to inherit the current journaling state of the directory. When this attribute is turned on for a journalized directory, all stream files, directories, and symbolic links that are created or linked into the directory (by adding a hard link or by renaming or moving the object) will automatically have journaling started by the system.

Note: Inherit journaling attribute considerations:

- If you rename an object in the same directory it currently resides in, journaling is not started for the object, even if the directory has the inherit current journaling state attribute on.
 - When a directory is moved to a directory which has the inherit journaling attribute on, only that moved directory has journaling started for it if appropriate. The objects within that moved directory are not affected.
 - If an object is restored to a directory which has the inherit journaling attribute on, journaling is not started for that object if the object has ever been journalized.
 - When using the Apply Journalized Changes (APYJRNCHG) command, the current value of the inherit journaling attribute for any directories is not used. Instead, any objects which are created as part of the apply have journaling started or not based on what happened during the runtime activity which is being applied.
- Object names and complete path names are contained within several journal entries of integrated file system objects. Object names and path names are National Language Support (NLS)-enabled.
 - If the system ends abnormally, system initial program load (IPL) recovery is provided for journalized integrated file system objects.
 - The maximum write limit supported by the various write interfaces is 2 GB - 1. The maximum journal entry size if RCVSIZOPT (*MAXOPT2 or *MAXOPT3) is specified is 4 000 000 000 bytes. Otherwise, the maximum journal entry size is 15 761 440 bytes. If you journal your stream file and have any writes that exceed 15 761 440 bytes, you need to use the *MAXOPT2 or *MAXOPT3 support to prevent any errors from occurring.

For more information about the layout of various journal entries, there is a C language include file, `qp0ljrn.h`, shipped in member `QSYSINC/H (QP0LJRN)`, that contains details of the integrated file system journal entry specific data content and formats.

Related concepts

“Stream file” on page 16

A *stream file* is a randomly accessible sequence of bytes, with no further structure imposed by the system.

“Directory” on page 4

A *directory* is a special object that is used to locate objects by names that you specify. Each directory contains a list of objects that are attached to it. That list can include other directories.

“Symbolic link” on page 13

A *symbolic link*, which is also called a soft link, is a path name contained in a file.

Related tasks

“Starting journaling” on page 95

To start journaling, do these steps on an object through the iSeries Navigator.

“Ending journaling” on page 95

After journaling has started on an object and, for whatever reasons, you want to end journaling on this object, you can use the steps described in this topic.

“Changing journaling” on page 95

After journaling has started on an object and, for whatever reasons, you want to change the journal attributes on the object without having to end and restart journaling, you can use the Change Journalled Object (CHGJRNOBJ) command to change journalled objects.

Related information

Journal management

Journal entry information finder

Journalled operations

These operations are only journalled when the type of the object or link that the operation is using is a type that can also be journalled.

- Create an object.
- Add a link to an existing object.
- Unlink a link.
- Rename a link.
- Rename a file identifier.
- Move a link into or out of the directory.

The following journalled operations are specific to a stream file:

- Data write or clear
- File truncate/extend
- File data forced
- Save with storage freed

The following journalled operations apply to all journalled object types:

- Attribute changes (including security changes such as authorities and ownership)
- Open
- Close
- Start journaling
- Change Journalled Object (CHGJRNOBJ) command
- End journaling
- Start the Apply Journalled Changes (APYJRNCHG) command
- End the Apply Journalled Changes (APYJRNCHG) command
- Save
- Restore

Related information

Journal Management

Journal entry information finder

Special considerations for journal entries

Many journalled integrated file system operations internally use commitment control to form a single transaction from the multiple functions performed during the operations.

These journaled operations should not be considered complete unless the commitment control cycle has a Commit journal entry (Journal Code C, Type CM). Journaled operations that contain a Rollback journal entry (Journal Code C, Type RB) in the commitment control cycle are failed operations, and the journal entries within them should not be replayed or replicated.

Journaled integrated file system entries (Journal Code B) that use commitment control in this manner include:

- AA — Change Audit Value
- B0 — Begin Create
- B1 — Create Summary
- B2 — Add link
- B3 — Rename/Move
- B4 — Unlink (Parent Directory)
- B5 — Unlink (Link)
- B7 — Created object authority information
- FA — Attribute Change
- JT — Start Journal (only when journaling is started because of an operation in a directory with the inherit journaling attribute of Yes)
- OA — Authority Change
- OG — Object Primary Group Change
- OO — Object Owner Change

Several integrated file system journal entries have a specific data field indicating whether the entry is a summary entry. Operations that send summary entry types will send two of the same entry types to the journal. The first entry contains a subset of the entry specific data. The second entry contains complete entry specific data and will indicate that it is a summary entry. Programs that are replicating the object or replaying the operation will generally only be interested in the summary entries.

For a create operation in a journaled directory, the B1 journal entry (Create Summary) is considered the summary entry.

Some journaled operations need to send a journal entry that is conversely related to the operation. For example, a commitment control cycle containing a B4 journal entry (Unlink) may also contain a B2 journal entry (Add Link). This type of scenario will only occur in operations that result in a Rollback journal entry (C — RB) .

This scenario may occur for two reasons:

1. The operation was about to fail, and the entry was needed internally for error path cleanup.
2. The operation was interrupted by a system outage, and during the subsequent IPL, recovery that needed to send the entry was performed to rollback the interrupted operation.

Related information

Journal entry information finder

Considerations for multiple hard links and journaling

If you have multiple hard links to a journaled integrated file system object, all the links should be saved and restored together so that the linkage is preserved as well as the associated journal information.

If specifying names in some of the journal related commands and if the names are really multiple hard links, then the object will only be operated on 'once'. The other hard links are essentially ignored.

Because multiple hard links point to the same object, and the journal entry has only the file identifier (File ID), which is the same for the object, then any journal interfaces that show the path name, for

example, Display Journal (DSPJRN), show only one link name for the object. However, this should not cause problems because one can operate on an object by any name and get the same result.

Related concepts

“Hard link” on page 12

A *hard link*, which is sometimes just called a link, cannot exist unless it is linked to an actual object.

Starting journaling

To start journaling, do these steps on an object through the iSeries Navigator.

1. Expand your system in **iSeries Navigator**.
2. Expand **File Systems**.
3. Right-click the object that you want to journal, and select **Journaling**.
4. After selecting the appropriate journaling options, click **Start**.

Results

To start journaling on an object through the character-based interface, you can use either the Start Journal (STRJRN) command or the QjoStartJournal API.

Related information

Start Journal (STRJRN) command

Start Journal (QjoStartJournal) API

Journal management

Changing journaling

After journaling has started on an object and, for whatever reasons, you want to change the journal attributes on the object without having to end and restart journaling, you can use the Change Journalized Object (CHGJRNOBJ) command to change journalized objects.

Related tasks

“Starting journaling”

To start journaling, do these steps on an object through the iSeries Navigator.

“Ending journaling”

After journaling has started on an object and, for whatever reasons, you want to end journaling on this object, you can use the steps described in this topic.

Related information

Change Journalized Object (CHGJRNOBJ) command

Ending journaling

After journaling has started on an object and, for whatever reasons, you want to end journaling on this object, you can use the steps described in this topic.

About this task

To end journaling on an object through iSeries Navigator, follow these steps:

1. Expand your system in **iSeries Navigator**.
2. Expand **File Systems**.
3. Right-click the object that you want to stop journaling, and select **Journaling**.
4. Click **End**.

Results

To end journaling on an object through the character-based interface, you can use either the End Journal (ENDJRN) command or the QjoEndJournal API.

Related tasks

“Starting journaling” on page 95

To start journaling, do these steps on an object through the iSeries Navigator.

Related information

End Journal (ENDJRN) command

End Journal (QjoEndJournal) API

Journal management

Reclaim operation of the “root” (/), QOpenSys, and user-defined file systems

Reclaiming the “root” (/), QOpenSys, and user-defined file systems can be accomplished using the Reclaim Object Links (RCLLNK) and Reclaim Storage (RCLSTG) commands.

By using the RCLLNK and RCLSTG commands, you can perform the following tasks:

- Correct object user profile problems
- Correct user-defined file system problems
- Correct internal object problems
- Remove invalid object links
- Handle damaged objects
- Create missing system objects
- Correct internal file system problems (RCLSTG only)
- Find lost objects (RCLSTG only)

Reclaim Object Links (RCLLNK) and Reclaim Storage (RCLSTG) commands comparison

You can use both the Reclaim Object Links (RCLLNK) and Reclaim Storage (RCLSTG) commands to correct problems in the “root” (/), QOpenSys, and user-defined file systems.

The RCLLNK command identifies and, if possible, corrects problems in mounted file systems that are in use. The RCLSTG command does not have this function. However, the RCLSTG command can correct problems that the RCLLNK command is unable to identify or correct. The following table provides a more detailed comparison between the two commands.

Table 11. RCLLNK and RCLSTG command comparison

	RCLLNK OBJ('/MyDir/MyObj')	RCLSTG ASPDEV(*SYSBAS)	RCLSTG ASPDEV(<IASPNAME>)
Is the system required to be in a restricted state?	No	Yes	No
Are all file systems usable during the reclaim operation?	Yes	No	The file systems in the independent ASP being reclaimed are unusable.
In what ASPs can objects be reclaimed?	Reclaims objects in system, user, and independent ASPs.	Reclaims objects in system and user ASPs.	Reclaims objects in independent ASPs.
How are objects reclaimed?	Objects are reclaimed on an individual or subtree basis as specified on the command.	Objects are reclaimed on a system-wide basis.	Objects are reclaimed on an independent ASP basis.

Table 11. RCLLNK and RCLSTG command comparison (continued)

	RCLLNK OBJ('/MyDir/MyObj')	RCLSTG ASPDEV(*SYSBAS)	RCLSTG ASPDEV(<IASPNAME>)
What known and applicable file system problems are identified and corrected, if possible?	Most (See "Reclaim operation of the "root" (/), QOpenSys, and user-defined file systems" on page 96 for more information.)	All	All
Are lost objects found?	No	Yes	Yes
Are objects in unmounted file systems reclaimed?	No	Yes	Yes
Is the command threadsafe?	Yes	No	No
How many instances of the command can be performed at the same time?	Multiple instances	Single instance	Single instance
What applicable integrated file system provided objects are re-created if necessary?	All	Most (See "Re-creation of integrated file system provided objects" on page 98 for more information.)	None
Can damaged objects be identified without being reclaimed?	Yes	No	No

Related concepts

"Examples: Reclaim Object Links (RCLLNK) command" on page 98

These examples describe situations in which the Reclaim Object Links (RCLLNK) command can be used to reclaim objects in the "root" (/), QOpenSys, and mounted user-defined file systems.

Related reference

"Re-creation of integrated file system provided objects" on page 98

This table shows the objects provided by the integrated file system that the Reclaim Object Links (RCLLNK) command re-creates if they do not exist. These objects are normally created during the initial program load (IPL). You can also re-create some of these objects, if necessary, using the Reclaim Storage (RCLSTG) command.

Related information

Reclaim Storage (RCLSTG) command

Reclaim Object Links (RCLLNK) command

Reclaim Object Links (RCLLNK) command

The Reclaim Object Links (RCLLNK) command identifies and repairs damaged objects in the "root" (/), QOpenSys, and mounted user-defined file systems without requiring the system to be in a restricted state. You can correct problems in these file systems without sacrificing productivity.

The RCLLNK command can be used as an alternative to the Reclaim Storage (RCLSTG) command in many situations. For example, RCLLNK is ideal for identifying and correcting problems in the following situations:

- Problems are isolated to a single object.
- Problems are isolated to a group of objects.
- Damaged objects need to be identified or deleted.
- The system cannot be in a restricted state during the reclaim operation.
- Independent ASPs must be available during the reclaim operation.

Re-creation of integrated file system provided objects

This table shows the objects provided by the integrated file system that the Reclaim Object Links (RCLLNK) command re-creates if they do not exist. These objects are normally created during the initial program load (IPL). You can also re-create some of these objects, if necessary, using the Reclaim Storage (RCLSTG) command.

Table 12. Objects provided by the integrated file system and re-created by the RCLLNK and RCLSTG commands

Path name	Type	Recreated by RCLLNK	Recreated by RCLSTG ASPDEV(*SYSBASE)
/dev/zero	*CHRFS	Yes	Yes
/dev/null	*CHRFS	Yes	Yes
/dev/xti/tcp	*CHRFS	Yes	No
/dev/xti/udp	*CHRFS	Yes	No
/etc/vfs	*STMF	Yes	No

In order for the RCLLNK command to re-create an object provided by the integrated file system that does not exist, it must be run with the SUBTREE parameter set to *DIR or *ALL while specifying the parent directory. The command must successfully reclaim the parent directory of the system object. For example, RCLLNK OBJ('/dev') SUBTREE(*DIR)

re-creates the /dev/zero and /dev/null *CHRFS objects if they do not exist.

In order for the RCLSTG command to re-create an integrated file system provided object that does not exist, it must be run with the ASPDEV parameter set to *SYSBASE and the directory recovery portion of reclaim must not be omitted.

Related concepts

“Provided directories” on page 7

The integrated file system creates these directories when the system is restarted if they do not already exist.

Related information

Reclaim Object Links (RCLLNK) command

Examples: Reclaim Object Links (RCLLNK) command

These examples describe situations in which the Reclaim Object Links (RCLLNK) command can be used to reclaim objects in the “root” (/), QOpenSys, and mounted user-defined file systems.

Example: Correcting problems for an object

In this situation, the known problems are isolated to one object. The object is damaged and unusable, and you cannot restore a backup version of the object from media. You need to correct the problem quickly without disrupting normal file system operations.

To reclaim the object, use this command:

```
RCLLNK OBJ('/MyDir/MyBadObject') SUBTREE(*NONE)
```

where /MyDir/MyBadObject is the name of the damaged and unusable object.

Example: Correcting problems that exist in a directory subtree

In this situation, the known problems are isolated to a group of objects within a directory subtree. An application is failing due to the problems within the directory subtree. You need to correct the problems quickly without disrupting normal file system operations.

| To reclaim the objects within the directory subtree, use this command:

```
| RCLLNK OBJ('/MyApplicationInstallDirectory') SUBTREE(*ALL)
```

| where MyApplicationInstallDirectory is the name of the directory containing the problem objects.

| **Example: Finding all damaged objects in the "root" (/), QOpenSys, and mounted user-defined file systems**

| In this situation, a disk failure has caused damage to a number of objects. You must identify the damaged objects before determining how to properly recover them.

| You need a solution to identify the damaged objects, but not take action against them. You must not disrupt normal file system operations.

| To identify the damaged objects, use this command:

```
| RCLLNK OBJ('/') SUBTREE(*ALL) DMGOBJOPT(*KEEP *KEEP)
```

| In addition, this command will also correct problems other than damaged objects as it identifies damaged objects.

| **Example: Deleting all damaged objects in the "root" (/), QOpenSys, and mounted user-defined file systems**

| In this situation, a disk failure caused a number of objects to become damaged. You must delete the damaged objects so that a backup copy of the objects can be restored from media.

| To delete the damaged objects, use this command:

```
| RCLLNK OBJ('/') SUBTREE(*ALL) DMGOBJOPT(*DELETE *DELETE)
```

| The damaged objects are deleted without disruption to normal file system operations. In addition, problems other than damage are corrected as the damaged objects are being deleted.

| **Example: Running multiple RCLLNK commands to quickly reclaim all objects in the "root" (/), QOpenSys, and mounted user-defined file systems**

| In this situation, as part of routine system maintenance, all objects in the "root" (/), QOpenSys, and mounted user-defined file systems are reclaimed. You want to finish the reclaim operation as quickly as possible to allow for additional system maintenance to be completed.

| By breaking the reclaim operation into separate groups, multiple RCLLNK commands can be performed concurrently to allow the reclaim operation to finish sooner.

| To run multiple reclaim operations against key system directories and other top-level directories, use the following commands (each in a separate job or thread).

```
| RCLLNK OBJ('/') SUBTREE(*DIR)
| RCLLNK OBJ('/tmp') SUBTREE(*ALL)
| RCLLNK OBJ('/home') SUBTREE(*ALL)
| RCLLNK OBJ('/etc') SUBTREE(*ALL)
| RCLLNK OBJ('/usr') SUBTREE(*ALL)
| RCLLNK OBJ('/QIBM') SUBTREE(*ALL)
| RCLLNK OBJ('/QOpenSys') SUBTREE(*ALL)
| RCLLNK OBJ('/IaspName') SUBTREE(*ALL)
| RCLLNK OBJ('/dev') SUBTREE(*ALL)
| RCLLNK OBJ('/OtherTopLevelDirectories') SUBTREE(*ALL)
```

| where OtherTopLevelDirectories are other directories that you want to reclaim.

Programming support

To take advantage of the stream files, directories, and other support of the integrated file system, you need to use a set of application programming interfaces (APIs) provided for accessing integrated file system functions.

Additionally, the addition of the integrated file system allows you to copy data between physical database files and stream files. You can perform this copy using CL commands, the data transfer function of iSeries Access Family, or APIs.

Copying data between stream files and database files

If you are familiar with operating on database files using record-oriented facilities such as data description specifications (DDS), you might find some fundamental differences in the way you operate on stream files.

The differences result from the different structure (or perhaps lack of structure) of stream files in comparison with database files. To access data in a stream file, you indicate a byte offset and a length. To access data in a database file, you typically define the fields to be used and the number of records to be processed.

Because you define the format and characteristics of a record-oriented file ahead of time, the operating system has knowledge of the file and can help you avoid performing operations that are not appropriate for the file format and characteristics. With stream files, the operating system has little or no knowledge of the format of the file. The application must know what the file looks like and how to operate on it properly. Stream files allow an extremely flexible programming environment, but at the cost of having little or no help from the operating system. Stream files are better suited for some programming situations; record-oriented files are better suited for other programming situations.

Related concepts

“Stream file” on page 16

A *stream file* is a randomly accessible sequence of bytes, with no further structure imposed by the system.

Copying data using CL commands

There are two sets of CL commands that allow you to copy data between stream files and database file members.

CPYTOSTMF and CPYFRMSTMF commands

You can use the Copy from Stream File (CPYFRMSTMF) and Copy to Stream File (CPYTOSTMF) commands to copy data between stream files and database file members. You can create a stream file from a database file member by using the CPYTOSTMF command. You can also create a database file member from a stream file by using the CPYFRMSTMF command. If the file or member that is the target of the copy does not exist, it is created.

There are some limitations, however. The database file must be either a program-described physical file containing only one field or a source physical file containing only one text field. The commands give you a variety of options for converting and reformatting the data that is being copied.

The CPYTOSTMF and CPYFRMSTMF commands can also be used to copy data between a stream file and a save file.

CPYTOIMPF and CPYFRMIMPF commands

You can also use the Copy to Import File (CPYTOIMPF) and Copy from Import File (CPYFRMIMPF) commands to copy data between stream files and database members. The CPYTOSTMF and

CPYFRMSTMF commands do not allow you to move data from complex, externally-described (DDS-described) database files. The word *import file* refers to the stream type file; the term typically refers to a file created for purposes of copying data between heterogeneous databases.

When copying from a stream (or import) file, the CPYFRMIMPF command allows you to specify a field definition file (FDF), which describes the data in the stream file. Or, you can specify that the stream files is delimited, and what characters are used to mark string, field, and record boundaries. Options for converting special data types such as time and date are also provided.

Data conversion is provided on these commands if the target stream file or database member already exists. If the file does not exist, you can use the following two-step method to get the data converted:

1. Use the CPYTOIMPF and CPYFRMIMPF commands to copy the data between the externally-described file and a source physical file.
2. Use the CPYTOSTMF and CPYFRMSTMF commands (which provide full data conversion regardless of whether the destination file exists) to copy between the source physical file and the stream file.

Here is an example:

```
| CPYTOIMPF FROMFILE(DB2FILE) TOFILE(EXPFILE) DTAFMT(*DLM)  
|          FLDDLML(';') RCDDLML('X'07') STRDLML(*DBLQUOTE) DATFMT(*USA) TIMFMT(*USA)
```

The DTAFMT parameter specifies that the input stream (import) file is delimited; the other choice is DTAFMT(*FIXED), which requires an field definition file to be specified. The FLDDLML, RCDDLML and STRDLML parameters identify the characters that act as the delimiters, or separators for fields, records, and strings.

The DATFMT and TIMFMT parameters indicate the format for any date and time information that is copied to the import file.

The commands are useful because they can be placed into a program, and they run entirely on your system. However, the interfaces are complex.

Related information

- Copy to Stream File (CPYTOSTMF) command
- Copy from Stream File (CPYFRMSTMF) command
- Copy to Import File (CPYTOIMPF) command
- Copy from Import File (CPYFRMIMPF) command
- Control language (CL)

Copying data using APIs

If you want to copy database file members to a stream file in an application, you can use the integrated file system open(), read(), and write() functions to open a member, read data from it, and write data to it or another file.

Related information

- open()--Open File API
- read()--Read from Descriptor API
- write()--Write to Descriptor API
- Integrated file system APIs

Copying data using data-transfer functions

The date-transfer applications in the iSeries Access Family licensed program have the advantage of an easy-to-follow graphical interface, and automatic numeric and character data conversion.

However, data transfer requires the installation of the iSeries Access Family product and requires the use of both PC and i5/OS resources and communications between the two.

If you have iSeries Access Family installed on the PC and your system, you can use the data transfer applications to transfer data between stream files and database files. You can also transfer data into a new database file that is based on an existing database file, into an externally described database file, or into a new database file definition and file.

Transferring data from a database file to a stream file:

To transfer a file from a database file to a stream file on your system, follow these steps.

1. Establish a connection to the system.
2. Map a network drive to the appropriate path in the i5/OS file system.
3. From the iSeries Access for Windows window, click **Data Transfer From iSeries server**.
4. Select the system you want to transfer from.
5. Select the file names, using the i5/OS database library and file name to copy from, and the network drive for the location of the resulting stream file. You can also click **PC File Details** to select the PC file format for the stream file. Data transfer supports common PC file types, such as ASCII text, BIFF3, CSV, DIF, Tab-delimited Text, or WK4.
6. Click **Transfer data from iSeries** to run the file transfer.

You can also perform this data movement in a batch job with the data transfer applications. Proceed as above, but select the **File** menu option to save the transfer request. The Data Transfer To iSeries server application creates a .DTT or a .TFR file. The Data Transfer From iSeries server application creates a .DTF or a .TTO file. In the iSeries Access Family directory, two programs can be run in batch from a command line:

- RTOPCB takes either a .DTF or a .TTO file as a parameter
- RFROMPCB takes either a .DTT or a .TFR file as a parameter

You can set either of these commands to run on a scheduled basis by using a scheduler application. For example, you can use the System Agent Tool (a part of the Microsoft® Plus Pack) to specify the program to run (for instance, RTOPCB MYFILE.TTO) and the time at which you want to run the program.

Transferring data from a stream file to a database file:

To transfer data from a stream file to a database file on your system, follow these steps.

1. Establish a connection to the system.
2. Map a network drive to the appropriate path in the i5/OS file system.
3. From the iSeries Access for Windows window, click **Data Transfer To iSeries server**.
4. Select the PC file name that you want to transfer. For the PC file name, you can choose **Browse** for the network drive you assigned, and choose a stream file. You can also use a stream file located on the PC itself.
5. Select the system on which you want the externally described database file to be located.
6. Click **Transfer data to iSeries server** to run the file transfer.

Note: If you are moving data to an existing database file definition on the system, the Data Transfer to iSeries server application requires you to use an associated format description file (FDF). A .FDF file describes the format of a stream file, and is created by the Data Transfer from iSeries server application when data is transferred from a database file to a stream file. To complete the transfer of data from a stream file to a database file, click **Transfer data to iSeries**. If an existing .FDF file is not available, you can quickly create a .FDF file.

You can also perform this data movement in a batch job with the data transfer applications. Proceed as above, but select the **File** menu option to save the transfer request. The Data Transfer To iSeries server

application creates a .DTT or a .TFR file. The Data Transfer From iSeries server application creates a .DTF or a .TTO file. In the iSeries Access Family directory, two programs can be run in batch from a command line:

- RTOPCB takes either a .DTF or a .TTO file as a parameter
- RFROMPCB takes either a .DTT or a .TFR file as a parameter

You can set either of these commands to run on a scheduled basis by using a scheduler application. For example, you can use the System Agent Tool (a part of the Microsoft Plus Pack) to specify the program to run (for instance, RTOPCB MYFILE.TTO) and the time at which you want to run the program.

Related reference

“Creating a format description file”

If you are moving data to an existing database file definition on the system, the Data Transfer to iSeries server application requires you to use an associated format description file (FDF).

Transferring data into a newly created database file definition and file:

You can follow these directions to transfer data into a newly created database file definition and file.

1. Establish a connection to the system.
2. Map a network drive to the appropriate path in the i5/OS file system.
3. From the iSeries Access for Windows window, click **Data Transfer to iSeries server**.
4. Open Tools of the Data Transfer to iSeries server application.
5. Click **Create iSeries database file**.

A wizard appears that allows you to create a new System i database file from an existing PC file. You need to specify the name of the PC file from which the System i file will be based, the name of the System i file to create, and several other necessary details. This tool parses a given stream file to determine the number, type, and size of the fields that are required in the resulting database file. The tool can then create the database file definition on your system.

Creating a format description file:

If you are moving data to an existing database file definition on the system, the Data Transfer to iSeries server application requires you to use an associated format description file (FDF).

An FDF file describes the format of a stream file, and is created by the Data Transfer from iSeries server application when data is transferred from a database file to a stream file.

To create a .FDF file:

1. Create an externally described database file with a format that matches your source stream file (number of fields, types of data).
2. Create one temporary data record within the database file.
3. Use the Data Transfer from iSeries server function to create a stream file and its associated .FDF file from this database file.

Now, you can use the Data Transfer to iSeries server function. Specify this .FDF file with the source stream file that you want to transfer.

Related reference

“Transferring data from a database file to a stream file” on page 102

To transfer a file from a database file to a stream file on your system, follow these steps.

“Transferring data from a stream file to a database file” on page 102

To transfer data from a stream file to a database file on your system, follow these steps.

Copying data between stream files and save files

A save file is used with save and restore commands to retain data that would otherwise be written on tape or diskette.

The file can also be used like a database file to read or write records that contain save/restore information. A save file can also be used to send objects to another user on the SNADS network.

You can use the Copy Object (CPY) command to copy a save file both to and from a stream file. However, when copying a stream file back into a save file object, the data must be valid save file data (it must have originated from a save file and been copied into a stream file).

By using a PC client, you can also access the save file and copy the data to your PC storage or LAN. Keep in mind, though, that data in save files cannot be accessed through the Network File System (NFS).

Related information

Copy Object (CPY) command

Performing operations using APIs

Many of the application programming interfaces (APIs) that perform operations on integrated file system objects are in the form of C language functions.

You have a choice of two sets of functions, either of which you can use in programs that are created using Integrated Language Environment (ILE) C:

- Integrated file system C functions that are included in the i5/OS operating system.
- C functions provided by the ILE C licensed program.

For information about the exit programs that the integrated file system supports, see Table 14 on page 109.

The integrated file system functions operate only through the integrated file system stream I/O support. The following APIs are supported:

Table 13. Integrated file system APIs

Function	Description
access()	Determine file accessibility
accessx()	Determine file accessibility for a class of users
chdir()	Change current directory
chmod()	Change file authorizations
chown()	Change owner and group of file
close()	Close file descriptor
closedir()	Close directory
creat()	Create new file or rewrite existing file
creat64()	Create new file or rewrite existing file (large file enabled)
DosSetFileLocks()	Lock and unlock byte range of a file
DosSetFileLocks64()	Lock and unlock byte range of a file (large file enabled)
DosSetRelMaxFH()	Change the maximum number of file descriptors
dup()	Duplicate open file descriptor
dup2()	Duplicate open file descriptor to another descriptor
faccessx()	Determine file accessibility for a class of users by descriptor

Table 13. Integrated file system APIs (continued)

Function	Description
fchdir()	Change current directory by descriptor
fchmod()	Change file authorizations by descriptor
fchown()	Change owner and group of file by descriptor
fclear()	Clear a file
fclear64()	Clear a file (large file enabled)
fcntl()	Perform file control action
fpathconf()	Get configurable path name variables by descriptor
fstat()	Get file information by descriptor
fstat64()	Get file information by descriptor (large file enabled)
fstatvfs()	Get information by descriptor
fstatvfs64()	Get information by descriptor (64-bit enabled)
fsync()	Synchronize changes to file
ftruncate()	Truncate file
ftruncate64()	Truncate file (large file enabled)
getcwd()	Get path name of current directory
getegid()	Get effective group ID
geteuid()	Get effective user ID
getgid()	Get real group ID
getgrgid()	Get group information using group ID
getgrnam()	Get group information using group name
getgroups()	Get group IDs
getpwnam()	Get user information for user name
getpwuid()	Get user information for user ID
getuid()	Get real user ID
givedescriptor()	Give file access to another job
ioctl()	Perform file I/O control action
link()	Create link to file
lseek()	Set file read/write offset
lseek64()	Set file read/write offset (large file enabled)
lstat()	Get file or link information
lstat64()	Get file or link information (large file enabled)
mkdir()	Make directory
mkfifo()	Make FIFO special file
mmap()	Create a memory map
mmap64()	Create a memory map (large file enabled)
mprotect()	Change a memory map protection
msync()	Synchronize a memory map
munmap()	Remove a memory map
open()	Open file
open64()	Open file (large file enabled)

Table 13. Integrated file system APIs (continued)

Function	Description
opendir()	Open directory
pathconf()	Get configurable path name variables
pread()	Read from descriptor with offset
pread64()	Read from descriptor with offset (large file enabled)
pwrite()	Write to descriptor with offset
pwrite64()	Write to descriptor with offset (large file enabled)
QjoEndJournal()	End journaling
QjoRetrieveJournal Information()	Retrieve journal information
QjoRetrieveJournalEntries()	Retrieve Journal Entries
QJORJIDI()	Retrieve journal identifier information
QJOSJRNE()	Send journal entry
QjoStartJournal()	Start journaling
QlgAccess()	Determine file accessibility (using NLS-enabled path name)
QlgAccessx()	Determine file accessibility for a class of users (using NLS-enabled path name)
QlgChdir()	Change current directory (using NLS-enabled path name)
QlgChmod()	Change file authorizations (using NLS-enabled path name)
QlgChown()	Change owner and group of file (using NLS-enabled path name)
QlgCreat()	Create new file or rewrite existing file (using NLS-enabled path name)
QlgCreat64()	Create new file or rewrite existing file (large file enabled and using NLS-enabled path name)
QlgCvtPathToQSYSObjName()	Resolve Integrated File System path name into QSYS Object Name (using NLS-enabled path name)
QlgGetAttr()	Get system attributes for an object (using NLS-enabled path name)
QlgGetcwd()	Get path name of current directory (using NLS-enabled path name)
QlgGetPathFromFileID()	Get path name of object from its file ID (using NLS-enabled path name)
QlgGetpwnam()	Get user information for user name (using NLS-enabled path name)
QlgGetpwnam_r()	Get user information for user name (using NLS-enabled path name)
QlgGetpwuid()	Get user information for user ID (using NLS-enabled path name)
QlgGetpwuid_r()	Get user information for user ID (using NLS-enabled path name)
QlgLchown()	Change owner and group of symbolic link (using NLS-enabled path name)
QlgLink()	Create link to file (using NLS-enabled path name)
QlgLstat()	Get file or link information (using NLS-enabled path name)

Table 13. Integrated file system APIs (continued)

Function	Description
QlgLstat64()	Get file or link information (large file enabled and using NLS-enabled path name)
QlgMkdir()	Make directory (using NLS-enabled path name)
QlgMkfifo()	Make FIFO special file (using NLS-enabled path name)
QlgOpen()	Open file (using NLS-enabled path name)
QlgOpen64()	Open file (large file enabled and using NLS-enabled path name)
QlgOpendir()	Open directory (using NLS-enabled path name)
QlgPathconf()	Get configurable path name variables (using NLS-enabled path name)
QlgProcessSubtree()	Process directories or objects within a directory tree (using NLS-enabled path name)
QlgReaddir()	Read directory entry (using NLS-enabled path name)
QlgReaddir_r()	Read directory entry (threadsafe and using NLS-enabled path name)
QlgReadlink()	Read value of symbolic link (using NLS-enabled path name)
QlgRenameKeep()	Rename file or directory, keep <i>new</i> if it exists (using NLS-enabled path name)
QlgRenameUnlink()	Rename file or directory, unlink <i>new</i> if it exists (using NLS-enabled path name)
QlgRmdir()	Remove directory (using NLS-enabled path name)
QlgSaveStgFree()	Save objects data and free its storage (using NLS-enabled path name)
QlgSetAttr()	Set system attributes for an object (using NLS-enabled path name)
QlgStat()	Get file information (using NLS-enabled path name)
QlgStat64()	Get file information (large file enabled and using NLS-enabled path name)
QlgStatvfs()	Get file system information (using NLS-enabled path name)
QlgStatvfs64()	Get file system information (large file enabled and using NLS-enabled path name)
QlgSymlink()	Make symbolic link (using NLS-enabled path name)
QlgUnlink()	Unlink file (using NLS-enabled path name)
QlgUtime()	Set file access and modification times (using NLS-enabled path name)
QP0FPTOS()	Perform miscellaneous file system functions
QP0LCHSG()	Change scan signature
Qp0ICvtPathToSYSObjName()	Resolve integrated file system path name into QSYS Object Name
QP0LFLOP()	Perform miscellaneous operations on objects
Qp0IGetAttr()	Get system attributes for an object
Qp0IGetPathFromFileID()	Get path name of object from its file ID
Qp0IOpen()	Open file with NLS-enabled path name
Qp0IProcessSubtree()	Process directories or objects within a directory tree

Table 13. Integrated file system APIs (continued)

Function	Description
Qp0lRenameKeep()	Rename file or directory, keep <i>new</i> if it exists
Qp0lRenameUnlink()	Rename file or directory, unlink <i>new</i> if it exists
QP0LROR()	Retrieve object references
QP0LRRO()	Retrieve referenced objects
QP0LRTSG()	Retrieve scan signature
Qp0lSaveStgFree()	Save objects data and free its storage
Qp0lSetAttr()	Set system attributes for an object
Qp0lUnlink()	Unlink file with NLS-enabled path name
Qp0zPipe()	Create interprocess channel with sockets
qsyssetgid()	Set effective group ID
qsyssetuid()	Set effective user ID
qsyssetgid()	Set group ID
qsyssetregid()	Set real and effective group IDs
qsyssetreuid()	Set real and effective user IDs
qsyssetuid()	Set user ID
QZNFRTVE()	Retrieve NFS export information
read()	Read from file
readdir()	Read directory entry
readdir_r()	Read directory entry (threadsafe)
readlink()	Read value of symbolic link
readv()	Read from file (vector)
rename()	Rename file or directory. Can be defined to have the semantics of Qp0lRenameKeep() or Qp0lRenameUnlink().
rewinddir()	Reset directory stream
rmdir()	Remove directory
select()	Check I/O status of multiple file descriptors
stat()	Get file information
stat64()	Get file information (large file enabled)
statvfs()	Get file system information
statvfs64()	Get file system information (large file enabled)
symlink()	Make symbolic link
sysconf()	Get system configuration variables
takedescriptor()	Take file access from another job
umask()	Set authorization mask for job
unlink()	Remove link to file
utime()	Set file access and modification times
write()	Write to file
writev()	Write to file (vector)

Note: Some of these functions are also used for i5/OS sockets.

Table 14. Integrated file system exit programs

Function	Description
Integrated File System Scan on Close API	Called during close processing such as with the close() API. This exit program must be provided by the user.
Integrated File System Scan on Open API	Called during open processing such as with the open() API. This exit program must be provided by the user.
Process a Path Name	Called by the Qp0lProcessSubtree() API for each object in the API's search that meets the caller's selection criteria. This exit program must be provided by the user.
Save Storage Free	Called by the Qp0lSaveStgFree() API to save an *STMF object type. This exit program must be provided by the user.

Related concepts

"File systems" on page 24

A *file system* provides you with the support to access specific segments of storage that are organized as logical units. These logical units on your system are files, directories, libraries, and objects.

Related reference

"Example: Integrated file system C functions" on page 115

This simple C language program illustrates the use of several integrated file system functions.

"Copying data using APIs" on page 101

If you want to copy database file members to a stream file in an application, you can use the integrated file system open(), read(), and write() functions to open a member, read data from it, and write data to it or another file.

Related information

Application programming interfaces (APIs)

ILE C functions

ILE C provides the standard C functions defined by the American National Standards Institute (ANSI).

These functions can operate through either the data management I/O support or the integrated file system stream I/O support, depending on what you specify when you create the C program. The compiler uses the data management I/O unless you tell it differently.

To tell the compiler to use the integrated file system stream I/O, you must specify *IFSIO for the System interface option (SYSIFCOPT) parameter in the Create ILE C Module (CRTCMOD) or Create Bound C Program (CRTBNDC) command. When you specify *IFSIO, the integrated file system I/O functions are bound instead of the data management I/O functions. In effect, the C functions of ILE C use the integrated file system functions to perform I/O.

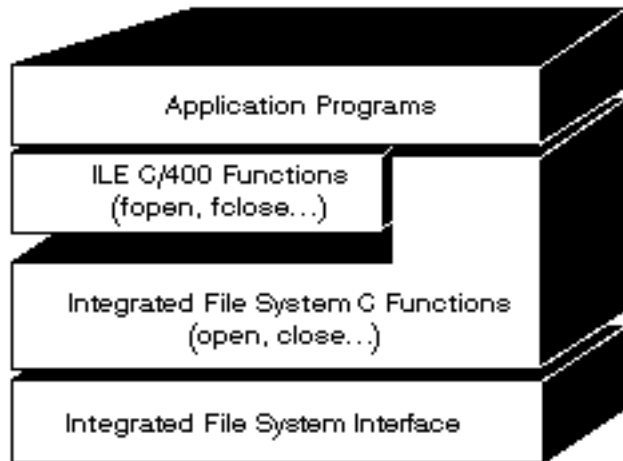




Figure 10. ILE C functions use the integrated file system stream I/O functions

For more information about using ILE C functions with integrated file system stream I/O, see the publication *WebSphere® Development Studio: ILE C/C++ Programmer's Guide* . For details on each C function of ILE C, see the publication *WebSphere Development Studio: C/C++ Language Reference* .

Large file support

The integrated file system APIs are enhanced to allow your applications to store and manipulate very large files. The integrated file system allows stream file sizes up to approximately 1 TB (1 TB equals approximately 1 099 511 627 776 bytes) in the “root” (/), QOpenSys, and user-defined file systems.

The integrated file system provides a set of 64-bit UNIX-type APIs and allows an easy mapping of existing 32-bit APIs to 64-bit APIs that are capable of accessing large file sizes and offsets by using eight byte integer arguments.

The following situations are provided to allow applications to use large file support:

- If the macro label `_LARGE_FILE_API` is defined at compile time, applications have access to APIs and data structures that are 64-bit enabled. For example, an application intending to use `stat64()` API and `stat64` structure will need to define `_LARGE_FILE_API` at compile time.
- If the macro label `_LARGE_FILES` is defined by the applications at compile time, existing APIs and data structures are mapped to their 64-bit versions. For example, if an application defines `_LARGE_FILES` at compile time, a call to `stat()` API is mapped to `stat64()` API and `stat()` structure is mapped to `stat64()` structure.

The applications that intend to use the large file support can either define `_LARGE_FILE_API` at compile time and code directly to the 64-bit APIs, or they can define `_LARGE_FILES` at compile time. All the appropriate APIs and data structures are then mapped to the 64-bit version automatically.

Applications that do not intend to use the large file support are not impacted and can continue to use integrated file system APIs without any changes.

Related information

Integrated file system APIs

`stat64()`--Get File Information (Large File Enabled) API

`stat()`--Get File Information API

Path name rules for APIs

When using an integrated file system or ILE C API to operate on an object, you identify the object by supplying its directory path. Here is a summary of rules to keep in mind when specifying path names in the APIs.

The term *object* in these rules refers to any directory, file, link, or other object.

- Path names are specified in hierarchical order beginning with the highest level of the directory hierarchy. The name of each component in the path is separated by a slash (/); for example:

```
Dir1/Dir2/Dir3/UsrFile
```

The backslash (\) is not recognized as a separator. It is handled as just another character in a name.

- Object names must be unique within a directory.
- The maximum length of each component of the path name and the maximum length of the path name string can vary for each file system.
- A / character at the beginning of a path name means that the path begins at the “root” (/) directory; for example:

```
/Dir1/Dir2/Dir3/UsrFile
```

- If the path name does not begin with a / character, the path is assumed to begin at the current directory; for example:

```
MyDir/MyFile
```

where MyDir is a subdirectory of the current directory.

- To avoid confusion with i5/OS special values, path names cannot start with a single asterisk (*) character. To specify a path name that begins with any number of characters, use two asterisks (**); for example:

```
'**.file'
```

This only applies to relative path names where no other characters precede before the asterisk (*).

- When operating on objects in the QSYS.LIB file system, the component names must be of the form *name.object-type*; for example:

```
/QSYS.LIB/PAYROLL.LIB/PAY.FILE
```

- When operating on objects in the independent ASP QSYS.LIB file system, the component names must be of the form *name.object-type*; for example:

```
'/asp_name/QSYS.LIB/PAYDAVE.LIB/PAY.FILE
```

- Do not use a colon (:) in path names. It has a special meaning within the system.
- Unlike path names in integrated file system commands, an asterisk (*), a question mark (?), a single quotation mark ('), a quotation mark ("), and a tilde (~) have no special significance. They are handled as if they are just another character in a name. To avoid confusion with i5/OS special values, path names should not start with a single asterisk (*) character. The only APIs that are exceptions to this rule are QjoEndJournal and QjoStartJournal.
- When using the Qlg (using NLS-enabled path names) API interfaces, a null character value is not allowed as one of the characters in the path name unless a null character is specified as a path name delimiter.

Related reference

“Path name rules for CL commands and displays” on page 68

When using an integrated file system command or display to operate on an object, you identify the object by supplying its path name.

Related information

End Journal (QjoEndJournal) API

Start Journal (QjoStartJournal) API

File descriptor

When using ILE C stream I/O functions as defined by the American National Standards Institute (ANSI) to perform operations on a file, you identify the file through the use of pointers. When using the integrated file system C functions, you identify the file by specifying a file descriptor. A *file descriptor* is a positive integer that must be unique in each job.

The job uses a file descriptor to identify an open file when performing operations on the file. The file descriptor is represented by the variable *filides* in C functions that operate on the integrated file system and by the variable *descriptor* in C functions that operate on sockets.

Each file descriptor refers to an *open file description*, which contains information such as a file offset, status of the file, and access modes for the file. The same open file description can be referred to by more than one file descriptor, but a file descriptor can refer to only one open file description.

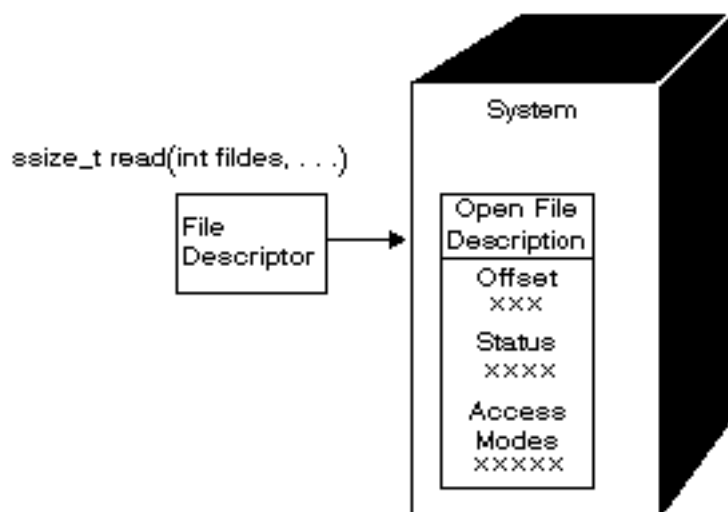


Figure 11. File descriptor and open file description

If an ILE C stream I/O function is used with the integrated file system, the ILE C runtime support converts the file pointer to a file descriptor.

When using the "root" (/), QOpenSys, or user-defined file systems, you can pass access to an open file description from one job to another, thus allowing the job to access the file. You do this by using the `givedescriptor()` or `takedescriptor()` function to pass the file descriptor between jobs.

Related information

`givedescriptor()`--Pass Descriptor Access to Another Job API

`takedescriptor()`--Receive Socket Access from Another Job API

Sockets programming


Sockets APIs

Security

When using the integrated file system APIs, you can restrict access to objects as you can when using data management interfaces. Be aware, however, that adopting authorities is not supported. An integrated file system API uses the authority of the user profile under which the job is running.

Each file system may have its own special authority requirements. NFS server jobs are the only exception to this rule. Network File System server requests run under the profile of the user whose user identification (UID) number was received by the NFS server at the time of the request.

Authorities on your system are the equivalent of *permissions* on UNIX systems. The types of permissions are read and write (for a file or a directory) and execute (for a file) or search (for a directory). The permissions are indicated by a set of permission bits, which make up the mode of access of the file or directory. You can change the permission bits by using the change mode functions `chmod()` or `fchmod()`. You can also use the `umask()` function to control which file permission bits are set each time a job creates a file.

For details on data security and authorities, see the publication Security Reference  .

Related information

`chmod()`--Change File Authorizations API

`fchmod()`--Change File Authorizations by Descriptor API

`umask()`--Set Authorization Mask for Job API

Integrated file system APIs

Socket support

If your application is using the “root” (/), QOpenSys, or user-defined file systems, you can take advantage of the integrated file system *local socket* support. A local socket object (object type *SOCKET) allows two jobs running on the same system to establish a communications connection with each other.

One of the jobs establishes a connection point by using the `bind()` C language function to create a local socket object. The other job specifies the name of the local socket object on the `connect()`, `sendto()`, or `sendmsg()` function.

After the connection is established, the two jobs can send data to and receive data from each other using the integrated file system functions such as `write()` and `read()`. None of the data that is transferred actually goes through the socket object. The socket object is just a meeting point where the two jobs can find each other.

When the two jobs are finished communicating, each job uses the `close()` function to close the socket connection. The local socket object remains in the system until it is removed using the `unlink()` function or the Remove Link (RMVLNK) command.

A local socket object cannot be saved.

Related information

Sockets programming

`write()`--Write to Descriptor API

`read()`--Read from Descriptor API

`close()`--Close File or Socket Descriptor API

`unlink()`--Remove Link to File API

Remove link (RMVLNK) command

Naming and international support

The support for the “root” (/) and QOpenSys file systems ensures that the characters in object names remain constant across encoding schemes used for different national languages and devices.

When an object name is passed to the system, each character of the name is converted to a 16-bit form in which all characters have a standard coded representation. When the name is used, it is converted to the appropriate coded-form for the code page being used.

If the code page to which the name is being converted does not contain a character used in a name, the name is rejected as not valid.

Because characters remain constant across code pages, you should not do an operation on the assumption that a particular character will change to another particular character when a specific code page is used. For example, you should not assume the number sign character will change to the pound sterling character even though they may have the same coded representation in different code pages.

Note that the names of the extended attributes of an object are converted in the same way as the name of the object, so the same considerations apply.

Related concepts

“Name continuity” on page 17

When you use the “root” (/), QOpenSys, and user-defined file systems, you can take advantage of system support that ensures characters in object names remain the same.

Data conversion

When you access files through the integrated file system, data in the files may or may not be converted, depending on the open mode requested when the file is opened.

An open file can be in one of two open modes:

Binary

The data is read from the file and written to the file without conversion. The application is responsible for handling the data.

Text

The data is read from the file and written to the file, assuming it is in textual form. When the data is read from the file, it is converted from the coded character set identifier (CCSID) of the file to the CCSID of the application, job, or system receiving the data. When data is written to the file, it is converted from the CCSID of the application, job, or system to the CCSID of the file. For true stream files, any line-formatting characters (such as carriage return, tab, and end-of-file) are just converted from one CCSID to another.

When reading from record files that are being used as stream files, end-of-line characters (carriage return and line feed) are appended to the end of the data in each record. When writing to record files:

- End-of-line characters are removed.
- Tab characters are replaced by the appropriate number of blanks to the next tab position.
- Lines are padded with either blanks (for a source physical file member) or nulls (for a data physical file member) to the end of the record.

On an open request, one of the following can be specified:

Binary, Forced

The data is processed as binary regardless of the actual content of the file. The application is responsible for knowing how to handle the data.

Text, Forced

The data is assumed to be text. The data is converted from the CCSID of the file to the CCSID of the application.

A default of *Binary, Forced* is used for the integrated file system `open()` function.

Related information

Example: Integrated file system C functions

This simple C language program illustrates the use of several integrated file system functions.

The program performs the following operations:

- 1 Uses the `getuid()` function to determine the real user ID (uid).
- 2 Uses the `getcwd()` function to determine the current directory.
- 3 Uses the `open()` function to create a file. It establishes read, write, and execute authority to the file for the owner (the person who created the file).
- 4 Uses the `write()` function to write a byte string to the file. The file descriptor that was provided in the open operation (3), identifies the file.
- 5 Uses the `close()` function to close the file.
- 6 Uses the `mkdir()` function to create a new subdirectory in the current directory. The owner is given read, write, and execute access to the subdirectory.
- 7 Uses the `chdir()` function to change the new subdirectory to the current directory.
- 8 Uses the `link()` function to create a link to the file that was previously created (3).
- 9 Uses the `open()` function to open the file for read only. The link that was created in (8) allows access to the file.
- 10 Uses the `read()` function to read a byte string from the file. The file descriptor that was provided in the open operation (9) identifies the file.
- 11 Uses the `close()` function to close the file.
- 12 Uses the `unlink()` function to remove the link to the file.
- 13 Uses the `chdir()` function to change the current directory back to the parent directory in which the new subdirectory was created.
- 14 Uses the `rmdir()` function to remove the subdirectory that was previously created (6).
- 15 Uses the `unlink()` function to remove the file that was previously created (3).

Note: This sample program will run correctly on systems where the CCSID of the job in which it is run is 37. The integrated file system APIs must have the object and path names encoded in the job's CCSID; however, the C compiler stores character constants in CCSID 37. For complete compatibility, translate character constants, such as object and path names, before passing APIs to the job's CCSID.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 132.

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>

#define BUFFER_SIZE      2048
#define NEW_DIRECTORY   "testdir"
#define TEST_FILE       "test.file"
#define TEST_DATA       "Hello World!"
#define USER_ID        "user_id_"
#define PARENT_DIRECTORY ".."
```

```

char InitialFile[BUFFER_SIZE];
char LinkName[BUFFER_SIZE];
char InitialDirectory[BUFFER_SIZE] = ".";
char Buffer[32];
int FilDes = -1;
int BytesRead;
int BytesWritten;
uid_t UserID;

void CleanUpOnError(int level)
{
    printf("Error encountered, cleaning up.\n");
    switch ( level )
    {
        case 1:
            printf("Could not get current working directory.\n");
            break;
        case 2:
            printf("Could not create file %s.\n",TEST_FILE);
            break;
        case 3:
            printf("Could not write to file %s.\n",TEST_FILE);
            close(FilDes);
            unlink(TEST_FILE);
            break;
        case 4:
            printf("Could not close file %s.\n",TEST_FILE);
            close(FilDes);
            unlink(TEST_FILE);
            break;
        case 5:
            printf("Could not make directory %s.\n",NEW_DIRECTORY);
            unlink(TEST_FILE);
            break;
        case 6:
            printf("Could not change to directory %s.\n",NEW_DIRECTORY);
            rmdir(NEW_DIRECTORY);
            unlink(TEST_FILE);
            break;
        case 7:
            printf("Could not create link %s to %s.\n",LinkName,InitialFile);
            chdir(PARENT_DIRECTORY);
            rmdir(NEW_DIRECTORY);
            unlink(TEST_FILE);
            break;
        case 8:
            printf("Could not open link %s.\n",LinkName);
            unlink(LinkName);
            chdir(PARENT_DIRECTORY);
            rmdir(NEW_DIRECTORY);
            unlink(TEST_FILE);
            break;
        case 9:
            printf("Could not read link %s.\n",LinkName);
            close(FilDes);
            unlink(LinkName);
            chdir(PARENT_DIRECTORY);
            rmdir(NEW_DIRECTORY);
            unlink(TEST_FILE);
            break;
        case 10:
            printf("Could not close link %s.\n",LinkName);
            close(FilDes);
            unlink(LinkName);
    }
}

```

```

        chdir(PARENT_DIRECTORY);
        rmdir(NEW_DIRECTORY);
        unlink(TEST_FILE);
        break;
    case 11:
        printf("Could not unlink link %s.\n",LinkName);
        unlink(LinkName);
        chdir(PARENT_DIRECTORY);
        rmdir(NEW_DIRECTORY);
        unlink(TEST_FILE);
        break;
    case 12:
        printf("Could not change to directory %s.\n",PARENT_DIRECTORY);
        chdir(PARENT_DIRECTORY);
        rmdir(NEW_DIRECTORY);
        unlink(TEST_FILE);
        break;
    case 13:
        printf("Could not remove directory %s.\n",NEW_DIRECTORY);
        rmdir(NEW_DIRECTORY);
        unlink(TEST_FILE);
        break;
    case 14:
        printf("Could not unlink file %s.\n",TEST_FILE);
        unlink(TEST_FILE);
        break;
    default:
        break;
}
printf("Program ended with Error.\n"\
      "All test files and directories may not have been removed.\n");
}

int main ()
{
    1
    /* Get and print the real user id with the getuid() function. */
    UserID = getuid();
    printf("The real user id is %u. \n",UserID);

    2
    /* Get the current working directory and store it in InitialDirectory. */
    if ( NULL == getcwd(InitialDirectory,BUFFER_SIZE) )
    {
        perror("getcwd Error");
        CleanUpOnError(1);
        return 0;
    }
    printf("The current working directory is %s. \n",InitialDirectory);

    3
    /* Create the file TEST_FILE for writing, if it does not exist.
       Give the owner authority to read, write, and execute. */
    FilDes = open(TEST_FILE, O_WRONLY | O_CREAT | O_EXCL, S_IRWXU);
    if ( -1 == FilDes )
    {
        perror("open Error");
        CleanUpOnError(2);
        return 0;
    }
    printf("Created %s in directory %s.\n",TEST_FILE,InitialDirectory);

    4
    /* Write TEST_DATA to TEST_FILE via FilDes */
    BytesWritten = write(FilDes,TEST_DATA,strlen(TEST_DATA));
    if ( -1 == BytesWritten )
    {

```

```

        perror("write Error");
        CleanUpOnError(3);
        return 0;
    }
    printf("Wrote %s to file %s.\n",TEST_DATA,TEST_FILE);

5
/* Close TEST_FILE via FilDes */
if ( -1 == close(FilDes) )
    {
        perror("close Error");
        CleanUpOnError(4);
        return 0;
    }
    FilDes = -1;
    printf("File %s closed.\n",TEST_FILE);

6
/* Make a new directory in the current working directory and
grant the owner read, write and execute authority */
if ( -1 == mkdir(NEW_DIRECTORY, S_IRWXU) )
    {
        perror("mkdir Error");
        CleanUpOnError(5);
        return 0;
    }
    printf("Created directory %s in directory %s.\n",NEW_DIRECTORY,InitialDirectory);

7
/* Change the current working directory to the
directory NEW_DIRECTORY just created. */
if ( -1 == chdir(NEW_DIRECTORY) )
    {
        perror("chdir Error");
        CleanUpOnError(6);
        return 0;
    }
    printf("Changed to directory %s/%s.\n",InitialDirectory,NEW_DIRECTORY);

/* Copy PARENT_DIRECTORY to InitialFile and
append "/" and TEST_FILE to InitialFile. */
strcpy(InitialFile,PARENT_DIRECTORY);
strcat(InitialFile,"/");
strcat(InitialFile,TEST_FILE);

/* Copy USER_ID to LinkName then append the
UserID as a string to LinkName. */
strcpy(LinkName, USER_ID);
sprintf(Buffer, "%d\0", (int)UserID);
strcat(LinkName, Buffer);

8
/* Create a link to the InitialFile name with the LinkName. */
if ( -1 == link(InitialFile,LinkName) )
    {
        perror("link Error");
        CleanUpOnError(7);
        return 0;
    }
    printf("Created a link %s to %s.\n",LinkName,InitialFile);

9
/* Open the LinkName file for reading only. */
if ( -1 == (FilDes = open(LinkName,O_RDONLY)) )
    {
        perror("open Error");
        CleanUpOnError(8);
    }

```

```

        return 0;
    }
    printf("Opened %s for reading.\n",LinkName);

10
/* Read from the LinkName file, via FilDes, into Buffer. */
BytesRead = read(FilDes,Buffer,sizeof(Buffer));
if ( -1 == BytesRead )
{
    perror("read Error");
    CleanupOnError(9);
    return 0;
}
printf("Read %s from %s.\n",Buffer,LinkName);
if ( BytesRead != BytesWritten )
{
    printf("WARNING: the number of bytes read is "\
          "not equal to the number of bytes written.\n");
}

11
/* Close the LinkName file via FilDes. */
if ( -1 == close(FilDes) )
{
    perror("close Error");
    CleanupOnError(10);
    return 0;
}
FilDes = -1;
printf("Closed %s.\n",LinkName);

12
/* Unlink the LinkName link to InitialFile. */
if ( -1 == unlink(LinkName) )
{
    perror("unlink Error");
    CleanupOnError(11);
    return 0;
}
printf("%s is unlinked.\n",LinkName);

13
/* Change the current working directory
back to the starting directory. */
if ( -1 == chdir(PARENT_DIRECTORY) )
{
    perror("chdir Error");
    CleanupOnError(12);
    return 0;
}
printf("changing directory to %s.\n",InitialDirectory);

14
/* Remove the directory NEW_DIRECTORY */
if ( -1 == rmdir(NEW_DIRECTORY) )
{
    perror("rmdir Error");
    CleanupOnError(13);
    return 0;
}
printf("Removing directory %s.\n",NEW_DIRECTORY);

15
/* Unlink the file TEST_FILE */
if ( -1 == unlink(TEST_FILE) )
{
    perror("unlink Error");
}

```

```

        CleanupOnError(14);
        return 0;
    }
    printf("Unlinking file %s.\n",TEST_FILE);

    printf("Program completed successfully.\n");
    return 0;
}

```

Working with files and folders using iSeries Navigator

You can perform these tasks with files and folders.

Checking in a file

To check in a file, follow these steps.

1. In **iSeries Navigator**, right-click the file that you want to check in.
2. Select **Properties**.
3. Select **File Properties** → **Use Page**.
4. Click **Check In**.

Checking out a file

To check out a file, follow these steps.

1. In **iSeries Navigator**, right-click the file that you want to check out.
2. Select **Properties**.
3. Click **File Properties** → **Use Page**.
4. Click **Check Out**.

Creating a folder

To create a folder, follow these steps.

1. Expand the system that you want to use in **iSeries Navigator** → **File Systems** → **Integrated File System**.
2. Right-click the file system to which you want to add the new folder and select **New Folder**.
3. Type a new name for the object in the **New Folder** dialog.
4. Click **OK**.

Results

When you create a folder on the System i platform, you need to consider whether you want to protect the new folder (or object) with journal management. You also need to consider whether you want objects created in this folder to be scanned or not.

Related tasks

“Setting whether objects should be scanned or not” on page 125
 Follow these steps to set whether an object should be scanned or not.

Related information

Journal management

Removing a folder

To remove a folder, follow these steps.

1. Expand the system that you want to use in **iSeries Navigator** → **File Systems** → **Integrated File System**. Continue to expand until the file or folder that you want to remove is visible.

2. Right-click the file or folder and select **Delete**.

Moving files or folders to another file system

Each file system has its own unique characteristics. However, moving objects to a different file system might mean losing the advantages of the file system in which the objects are currently stored. You might want to move objects from one file system to another to take advantage of those characteristics.

About this task

Before moving objects to another file system, you should be familiar with the file systems on the integrated file system and their characteristics.

You should also consider the following situation:

- Are you using applications that use advantages of the file system that the objects are currently in?
Some file systems support interfaces that are not part of the integrated file system support. Applications that use these interfaces may no longer be able to access objects that are moved to another file system. For example, the QDLS and QOPT file systems support the hierarchical file system (HFS) APIs and commands to work with document and folder objects. You cannot use these interfaces on objects that are in other file systems.
- What characteristics of the objects are important to you?
Not all characteristics are supported by all file systems. For example, the QSYS.LIB or independent ASP QSYS.LIB file systems support storing and retrieving only a few extended attributes, whereas the "root" (/) and QOpenSys file systems support storing and retrieving all extended attributes. Therefore QSYS.LIB and independent ASP QSYS.LIB are not good candidates for storing objects that have extended attributes.
Good candidates for moving are the PC files that are stored in QDLS. Most PC applications should be able to continue working with PC files that are moved from QDLS to other file systems. The "root" (/), QOpenSys, QNetWare, and QNTC file systems are good choices for storing these PC files. Because they support many of the OS/2 file system characteristics, these file systems can provide faster access to files.

To move objects to another file system, perform the following steps:

1. Save a copy of all objects that you are planning to move.
Having a backup copy allows you to restore the objects to the original file system if you find that applications cannot access the objects in the file system to which you have moved them.
Note: You cannot save objects from one file system and restore them to another.
2. Create the directories in the file system that you want to move the objects to using the Create Directory (CRTDIR) command.
You should carefully examine the attributes of the directory the objects are currently in to determine if you want to duplicate those attributes on the directories you create. For example, the user who creates the directory is its owner, rather than the user who owned the old directory. You may want to transfer ownership of the directory after you have created it, if the file system supports setting the owner of a directory.
3. Move the files to the file system that you have chosen using the Move Object (MOV) command.
MOV is recommended because it preserves the ownership of the objects, if the file system supports setting the ownership of objects. You can, however, use the Copy Object (CPY) command to preserve the ownership of the objects by using the OWNER(*KEEP) parameter. Keep in mind that this only works for file systems that support setting the owner of an object. Note that when using MOV or CPY:
 - Attributes may not match and may be discarded.
 - Extended attributes may be discarded.

- Authorities may not be equivalent and may be discarded.

This means that if you decide to return the object to its original file system, you may not want to just move or copy it back because of the attributes and authorities that have been discarded. The safest way to return an object is to restore a saved version of it.

Related concepts

“File systems” on page 24

A *file system* provides you with the support to access specific segments of storage that are organized as logical units. These logical units on your system are files, directories, libraries, and objects.

Related reference

“File system comparison” on page 25

These tables summarize the features and limitations of each file system.

Related information

Create Directory (CRTDIR) command

Move Object (MOV) command

Copy Object (COPY) command

Setting permissions

Adding permissions to an object allows you to control the ability of others to manipulate that object. With permissions, you can allow some users to only view objects, while allowing others to actually edit the objects.

About this task

To set permissions to a file or folder, follow these steps:

1. Expand the system that you want to use in **iSeries Navigator** → **File Systems** → **Integrated File System**. Continue to expand until the object for which you want to add permissions is visible.
2. Right-click the object for which you want to add permissions and select **Permissions**.
3. Click **Add** on the **Permissions** dialog.
4. Select one or more users and groups or enter the name of a user or group in the user or groups name field in the **Add** dialog.
5. Click **OK**. This will add the users or groups to the top of the list.
6. Click the **Details** button to implement detailed permissions.
7. Apply the permissions you want for the user by checking the box by the appropriate check box.
8. Click **OK**.

Setting up file text conversion

You can set up automatic text file conversion in iSeries Navigator. Automatic text file conversion allows you to use file extensions for file data conversion.

About this task

The integrated file system can convert a data file when it is transferred between a System i platform and a PC. When you access the data file from a PC, it is handled as if it were in ASCII.

To set up file text conversion, follow these steps:

1. Expand the system that you want to use in **iSeries Navigator** → **File Systems**.
2. Right-click **Integrated File System** and select **Properties**.
3. Enter the file extension that you want to convert automatically in the **File extensions for automatic text file conversion** text box and click **Add**.
4. Repeat step 3 for all file extensions that you want to convert automatically.

5. Click **OK**.

Results

Sending a file or folder to another system

To send a file or folder to another system, follow these steps.

1. Expand the system that you want to use in **iSeries Navigator** → **File Systems** → **Integrated File System**. Continue to expand until the file or folder that you want to send is visible.
2. Right-click the file or folder and select **Send**. The file or folder appears in the Selected Files and Folders list of the Send Files from dialog.
3. Expand the list of available systems and groups.
4. Select a system and click **Add** to add the system to the **Target systems and groups** list. Repeat this step for all the systems you want to send this file or folder.
5. Click **OK** to send the file or folder with the current default package definitions and schedule information.

Results

When you create a package definition, it is saved and can be reused at any time to send the defined set of files and folders to multiple endpoint systems or system groups. If you choose to create a snapshot of your files, you can keep more than one version of copies of the same set of files. Sending a snapshot ensures that no updates are made to the files during the distribution, so that the last target system receives the same objects as the first target system.

Related tasks

“Changing options for a package definition”

A package definition allows you to group together a set of i5/OS objects or integrated file system files.

“Scheduling a date and time to send your file or folder” on page 124

The schedule function gives you the flexibility to do your work when it is convenient for you to do it.

Changing options for a package definition

A package definition allows you to group together a set of i5/OS objects or integrated file system files.

About this task

The package definition also allows you to view this same group of files as a logical set, or as a physical set, by taking a snapshot of the files to preserve them for later distribution.

To change the options for the package definitions, follow these steps:

1. Complete the steps for “Sending a file or folder to another system.”
2. Click the **Options** tab. The default options are to include subfolders when packaging and sending files and to replace an existing file with the file being sent.
3. Change these options as required.
4. Click **Advanced** to set advanced save and restore options.
5. Click **OK** to save the advanced options.
6. Click **OK** to send the file, or click **Schedule** to set a time for sending the file.

Related tasks

“Scheduling a date and time to send your file or folder” on page 124

The schedule function gives you the flexibility to do your work when it is convenient for you to do it.

Scheduling a date and time to send your file or folder

The schedule function gives you the flexibility to do your work when it is convenient for you to do it.

About this task

To schedule a date and time to send your file or folder, follow these steps:

1. Complete the steps for “Sending a file or folder to another system” on page 123.
2. Click **Schedule**.
3. Select the options for when you want to send the file or folder.

Creating a file share

A *file share* is a directory path that iSeries NetServer shares with PC clients on the System i network. A file share can consist of any integrated file system directory on the System i platform.

About this task

To create a file share, follow these steps.

1. Expand your system in **iSeries Navigator** → **File Systems** → **Integrated File System**.
2. Expand the file system that contains the folder for which you want to create a share.
3. Right-click the folder for which you want to create a share and select **Sharing**.
4. Select **New Share**.

Changing a file share

A *file share* is a directory path that iSeries NetServer shares with PC clients on the System i network. A file share can consist of any integrated file system directory on the System i platform.

About this task

To change a file share, follow these steps.

1. Expand your system in **iSeries Navigator** → **File Systems** → **Integrated File System**.
2. Expand the folder that has the share defined for it that you want to change.
3. Right-click the folder that has the share defined for it that you want to **Sharing**.
4. Select **New Share**.

Creating a new user-defined file system

A user-defined file system (UDFS) is a file system that you create and define the attributes for. UDFSs reside in auxiliary storage pools (ASPs) on the system.

About this task

To create a new user-defined file system (UDFS), follow these steps:

1. Expand your system in **iSeries Navigator** → **File Systems** → **Integrated File System** → **Root** → **Dev**.
2. Click the auxiliary storage pool (ASP) that you want to contain the new UDFS.
3. Select **New UDFS** from the File menu.
4. Specify the UDFS name, description (optional), auditing values, default file format, default scanning attribute, and whether the files in the new UDFS will have case-sensitive file names on the New User-Defined File System dialog.

Mounting a user-defined file system

To access or view the data stored in a UDFS, you must mount the UDFS after every IPL.

About this task

When you mount a UDFS, it covers up any file systems, directories, or objects that exist beneath the mount point in the folder hierarchy. This makes those file systems, directories, or objects inaccessible until you unmount the UDFS. To ensure that access to all data in the integrated file system is maintained, mount the UDFS over an empty folder. After the UDFS is mounted, the files within the UDFS will be accessible from within that folder. Any changes made in the folder will be changes to the UDFS, rather than to the covered up folder.

Note: A UDFS on an independent ASP cannot be mounted over.

To mount a user-defined file system (UDFS), follow these steps:

1. Expand your system in **iSeries Navigator** → **File Systems** → **Integrated File System** → **Root** → **Dev**.
2. Click the auxiliary storage pool (ASP) that contains the UDFS that you want to mount.
3. Right-click the UDFS that you want to mount in the **UDFS Name** column of Operations Navigator's right pane.
4. Select **Mount**.

Results

If you like to drag, you can mount a UDFS by dragging it to a folder within the integrated file system on the same system. You cannot drop the UDFS on `/dev`, `/dev/QASPxx`, `/dev/asp_name`, another system, or the desktop.

Unmounting a user-defined file system

When you mount a UDFS, it covers up any file systems, directories, or objects that exist beneath the mount point in the folder hierarchy. This makes those file systems, directories, or objects inaccessible until you unmount the UDFS.

About this task

To unmount a user-defined file system (UDFS), follow these steps:

1. Expand your system in **iSeries Navigator** → **File Systems** → **Integrated File System** → **Root** → **Dev**.
2. Click the auxiliary storage pool (ASP) that contains the UDFS that you want to unmount.
3. Right-click the UDFS that you want to unmount in the **UDFS Name** column of iSeries Navigator's right pane.
4. Select **Unmount**.

Setting whether objects should be scanned or not

Follow these steps to set whether an object should be scanned or not.

1. Expand your system in **iSeries Navigator** → **File Systems** → **Integrated File System**.
2. Expand the folder or file of interest.
3. Right-click the folder or file and select **Properties**
4. Select the **Security** tab.
5. Select **Scan objects** with the option that you want.

Results

For more information about the options, see the following sections. The descriptions for these options are for files. Only files may be scanned. With folders and user-defined file systems, you can specify what scan attribute should be given to files created in that folder or user-defined file system.

- Yes

The object will be scanned according to the rules described in the scan-related exit programs if the object has been modified or if the scanning software has been updated since the last time the object was scanned.

- No

The object will not be scanned by the scan-related exit programs.

Note: If the Scan on next access after object has been restored option is selected in the system values when an object with this attribute is restored, the object will be scanned at least once after the restore.

- Only when the object has changed

The object will be scanned according to the rules described in the scan-related exit programs only if the object has been modified since the last time the object was scanned. It will not be scanned if the scanning software has been updated.

If the Use only when objects have changed attribute to control scan option is not specified in the system values, this object change only attribute will not be used, and the object will be scanned after it is modified and when scan software indicates an update.

Notes:

1. On this tab for files, you can also determine the scan status of an object.
2. If the Scan on next access after object has been restored option is selected in the system values when an object with this attribute is restored, the object will be scanned at least once after the restore.

Transport-independent remote procedure call

Developed by Sun Microsystems, remote procedure call (RPC) easily separates and distributes client applications from a server mechanism.

RPC includes a standard for data representation, called eXternal Data Representation (XDR), to enable more than one type of machine to access transmitted data. Transport-independent RPC (TI-RPC) is the latest version of RPC. It provides a method of separating the underlying protocol that is used at the network layer, providing a more seamless transition from one protocol to another. The only protocols that are currently available on the System i platform are TCP and UDP.

Developing distributed applications across a network is a seamless task when using RPC. The primary targets are applications that gravitate more toward distributing the user interface or data retrieval.

Network selection APIs

These APIs provide the means to choose the transport on which an application should run.

These APIs require that *STMF /etc/netconfig file exist on the system. If the netconfig file does not exist in the /etc directory, the user must copy it from /QIBM/ProdData/OS400/RPC directory. The netconfig file is always in the /QIBM/ProdData/OS400/RPC directory.

API	Description
endnetconfig()	Releases the pointer to the records stored in the netconfig file
freenetconfigent()	Frees the netconfig structure that is returned from the call to the getnetconfigent() function
getnetconfig()	Returns the pointer to the current record in the netconfig file and increments its pointer to the next record

API	Description
getnetconfig()	Returns the pointer to the netconfig structure that corresponds to the input netid
setnetconfig()	Initializes the record pointer to the first entry in the netconfig file. The setnetconfig() function must be used before the first use of getnetconfig() function. The setnetconfig() function returns a unique handle (a pointer to the records stored in the netconfig file) to be used by the getnetconfig() function.

Related information

API finder

Name-to-address translation APIs

These APIs allow an application to obtain the address of a service or a specified host in a transport-independent manner.

API	Description
netdir_free()	Frees structures that are allocated by name-to-address translation APIs
netdir_getbyaddr()	Maps addresses into host names and service names
netdir_getbyname()	Maps the host name and service name that are specified in the service parameter to a set of addresses that are consistent with the transport identified in the netconfig structure
netdir_options()	Provides interfaces to transport-specific capabilities such as the broadcast address and reserved port facilities of TCP and UDP
netdir_sperror()	Issues an informational message that states why one of the name-to-address translation APIs failed
taddr2uaddr()	Translates a transport-specific (local) address to a transport-independent (universal) address
uaddr2taddr()	Translates a transport-independent (universal) address to a transport-specific (local) address (netbuf structure)

Related information

API finder

eXternal Data Representation (XDR) APIs

These APIs allow RPC applications to handle arbitrary data structures, regardless of their different hosts' byte orders or structure layout conventions.

API	Description
xdr_array()	A filter primitive that translates between variable-length arrays and their corresponding external representations. This function is called to encode or decode each element of the array
xdr_bool()	A filter primitive that translates between Booleans (C integers) and their external representations. When encoding data, this filter produces values of either 1 or 0.
xdr_bytes()	A filter primitive that translates between counted byte arrays and their external representations. This function treats a subset of generic arrays in which the size of array elements is known to be 1 and the external description of each element is built-in. The length of the byte sequence is explicitly located in an unsigned integer. The byte sequence is not ended by a null character. The external representation of the bytes is the same as their internal representation.

API	Description
xdr_char()	A filter primitive that translates between C-language characters and their external representation
xdr_double()	A filter primitive that translates between C-language double-precision numbers and their external representations
xdr_double_char()	A filter primitive that translates between C-language 2-byte characters and their external representation
xdr_enum()	A filter primitive that translates between C-language enumeration (enum) and its external representation
xdr_free()	Recursively frees the object pointed to by the pointer passed in
xdr_float()	A filter primitive that translates between C-language floating-point numbers (normalized single floating-point numbers) and their external representations
xdr_int()	A filter primitive that translates between C-language integers and their external representation
xdr_long()	A filter primitive that translates between C-language long integers and their external representations
xdr_netobj()	A filter primitive that translates between variable-length opaque data and its external representation
xdr_opaque()	A filter primitive that translates between fixed-size opaque data and its external representation
xdr_pointer()	Provides pointer chasing within structures and serializes null pointers. Can represent recursive data structures, such as binary trees or linked lists.
xdr_reference()	a filter primitive that provides pointer chasing within structures. This primitive allows the serializing, deserializing, and freeing of any pointers within one structure that are referenced by another structure. The xdr_reference() function does not attach special meaning to a null pointer during serialization, and passing the address of a null pointer may cause a memory error. Therefore, the programmer must describe data with a two-sided discriminated union. One side is used when the pointer is valid; the other side, when the pointer is null.
xdr_short()	A filter primitive that translates between C-language short integers and their external representation
xdr_string()	A filter primitive that translates between C-language strings and their corresponding external representations
xdr_u_char()	A filter primitive that translates between unsigned C-language characters and their external representations
xdr_u_int()	A filter primitive that translates between C-language unsigned integers and their external representations
xdr_u_long()	A filter primitive that translates between C-language unsigned long integers and their external representations
xdr_u_short()	A filter primitive that translates between C-language unsigned short integers and their external representations
xdr_union()	A filter primitive that translates between discriminated C unions and their corresponding external representations
xdr_vector()	A filter primitive that translates between fixed-length arrays and their corresponding external representations
xdr_void()	Has no parameters. It is passed to other RPC functions that require a parameter, but does not transmit data

API	Description
xdr_wrapstring()	A primitive that calls the xdr_string(xdr, sp, maxuint) API, where maxuint is the maximum value of an unsigned integer. The xdr_wrapstring() is useful because the RPC package passes a maximum of two XDR functions as parameters, and the xdr_string() function requires three.

Related information

API finder

Authentication APIs

These APIs provide authentication to the TI-RPC applications.

API	Description
auth_destroy()	Destroys the authentication information structure that is pointed to by the auth parameter
authnone_create()	Creates and returns a default RPC authentication handle that passes null authentication information with each remote procedure call.
authsys_create()	Creates and returns an RPC authentication handle that contains authentication information

Related information

API finder

Transport-independent RPC (TI-RPC) APIs

These APIs provide a distributed application development environment by isolating the application from any specific transport feature. This adds ease-of-use to the transports.

Related information

API finder

TI-RPC simplified APIs

These simplified APIs specify the type of transport to use. Applications using this level do not have to explicitly create handles.

API	Description
rpc_call()	Call a remote procedure on the specified system
rpc_reg()	Register a procedure with RPC service package

Related information

API finder

TI-RPC top-level APIs

These APIs allow the application to specify the type of transport.

API	Description
clnt_call()	Call a remote procedure associated with the client
clnt_control()	Change information about a client object
clnt_create()	Create a generic client handle
clnt_destroy()	Destroy the client's RPC handle
svc_create()	Create a server handle
svc_destroy()	Destroy an RPC service transport handle

Related information

API finder

TI-RPC intermediate-level APIs

These APIs are similar to the top-level APIs, but the user applications select the transport specific information using network selection APIs.

API	Description
clnt_tp_create()	Create a client handle
svc_tp_create()	Create a server handle

Related information

API finder

TI-RPC expert-level APIs

These APIs allow the application to select which transport to use. They also offer an increased level of control over the details of the CLIENT and SVCXPRT handles. These APIs are similar to the intermediate-level APIs with an additional control that is provided by using the name-to-address translation APIs.

An additional control that is provided by using the name-to-address translation APIs.

API	Description
clnt_tli_create()	Create a client handle
rpcb_getaddr()	Find the universal address of a service
rpcb_set()	Register the server address with the RPCbind
rpcb_unset()	Used by the servers to unregister their addresses
svc_reg()	Associate program and version with dispatch
svc_tli_create()	Create a server handle
svc_unreg()	Delete an association set by svc_reg()

Related information

API finder

Other TI-RPC APIs

These APIs allow the various applications to work in coordination with the simplified, top-level, intermediate-level, and expert-level APIs.

API	Description
clnt_freeres()	Free data allocated by the RPC or XDR system
clnt_geterr()	Get the error structure from the client handle
svc_freeargs()	Free data allocated by the RPC or XDR system
svc_getargs()	Decodes the arguments of an RPC request
svc_getrpccaller()	Get the network address of the caller
svc_run()	Waits for RPC requests to arrive
svc_sendreply()	Sends the results of a procedure call to a remote client.
svcerr_decode()	Sends information to client for decode error
svcerr_noproc()	Sends information to client for procedure number error

API	Description
svcerr_systemerr()	Sends information to client for system error

Related information

API finder

Related information for integrated file system

Listed here are the product manuals, Web sites, and information center topics that relate to the integrated file system topic. You can view or print any of the PDFs.

Manuals

- [i5/OS Network File System Support !\[\]\(79de0df6c6ddd2d4eb74f1cc5f48ec50_img.jpg\)](#). This book describes the Network File System through a series of real-life applications. Included is information about exporting, mounting, file locking, and security considerations. From this book, you can learn how to use NFS to construct and develop a secure network namespace.
- [WebSphere Development Studio: ILE C/C++ Language Reference !\[\]\(d4c9768318b38eff1042b07478e20b4c_img.jpg\)](#). This book provides information needed to design, edit, compile, run, and debug ILE C programs on the System i platform.
- [Security Reference !\[\]\(27d314856359a9d7feca17161bc1f4a4_img.jpg\)](#). This book provides detailed technical information about i5/OS security including security related system values that impact integrated file system scan-related processing.
- [APPC Programming !\[\]\(d355663486c698e3972a8b93ac8b2102_img.jpg\)](#). This book describes the advanced program-to-program communications (APPC) support for System i platforms. It guides in developing application programs that use APPC and in defining the communications environment for APPC.
- [Backup and Recovery !\[\]\(1858f6a9022d088c0a7eca873f99643b_img.jpg\)](#). This book provides general information about recovery and availability options for System i platforms.

Other information

- **Experience reports**

Experience reports are written by IBM® developers, documenting their hands-on experiences implementing real-world scenarios and solutions. Use them to follow the experiences of IBM developers with a specific implementation of a System i solution, complete with step-by-step instructions and tips. The experience report Backing up the integrated file system is related to files and file systems.


- Control language
- i5/OS globalization
- Application programming interfaces
- Journal management
- Commitment control

Saving PDF files

To save a PDF on your workstation for viewing or printing:

1. Right-click the PDF in your browser (right-click the link above).
2. Click the option that saves the PDF locally.
3. Navigate to the directory in which you want to save the PDF.
4. Click **Save**.

Downloading Adobe Reader

- | You need Adobe Reader installed on your system to view or print these PDFs. You can download a free copy from the Adobe Web site (www.adobe.com/products/acrobat/readstep.html) .

Code license and disclaimer information

IBM grants you a nonexclusive copyright license to use all programming code examples from which you can generate similar function tailored to your own specific needs.

- | SUBJECT TO ANY STATUTORY WARRANTIES WHICH CANNOT BE EXCLUDED, IBM, ITS PROGRAM DEVELOPERS AND SUPPLIERS MAKE NO WARRANTIES OR CONDITIONS EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT, REGARDING THE PROGRAM OR TECHNICAL SUPPORT, IF ANY.
- | UNDER NO CIRCUMSTANCES IS IBM, ITS PROGRAM DEVELOPERS OR SUPPLIERS LIABLE FOR ANY OF THE FOLLOWING, EVEN IF INFORMED OF THEIR POSSIBILITY:
 - | 1. LOSS OF, OR DAMAGE TO, DATA;
 - | 2. DIRECT, SPECIAL, INCIDENTAL, OR INDIRECT DAMAGES, OR FOR ANY ECONOMIC CONSEQUENTIAL DAMAGES; OR
 - | 3. LOST PROFITS, BUSINESS, REVENUE, GOODWILL, OR ANTICIPATED SAVINGS.
- | SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF DIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, SO SOME OR ALL OF THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

Appendix. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

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

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation

Software Interoperability Coordinator, Department YBWA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

- | The licensed program described in this information and all licensed material available for it are provided
- | by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement,
- | IBM License Agreement for Machine Code, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming Interface Information

This Integrated file system publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM i5/OS.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

- | DB2
- | i5/OS
- | IBM
- | IBM (logo)
- | Integrated Language Environment
- | iSeries
- | NetServer
- | OfficeVision
- | OS/2
- | OS/400
- | System i
- | System x
- | WebSphere
- | xSeries

Microsoft, Windows, Windows NT and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

- | Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Terms and conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

Personal Use: You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of these publications, or any portion thereof, without the express consent of IBM.

Commercial Use: You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF

MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.



Printed in USA