IBM

@server

iSeries

Integrated File System Introduction

Version 5

IBM

@server

iSeries

Integrated File System Introduction

Version 5

Contents

About Integrated File System Introduction

This book provides an overview of the integrated file system, which includes:
* What is the integrated file system?
* Why you might want to use it.
* Integrated file system concepts and terminology.
* The interfaces you can use to interact with the integrated file system.
* The APIs and techniques you can use to create programs that interact with the integrated file system.
* Characteristics of individual file systems.

Who should read the Integrated File System Introduction book

This book is intended for iSeries server users, programmers, and managers who want to understand the integrated file system and how to use it.

Code disclaimer information

This document contains programming examples.

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.

All sample code is provided by IBM for illustrative purposes only. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

All programs contained herein are provided to you ″AS IS″ without any warranties of any kind. The implied warranties of non-infringement, merchantability and fitness for a particular purpose are expressly disclaimed.

Chapter 1. Introduction to the integrated file system

The following topics describe the integrated file system on your iSeries server and show how it can be of use on your server.

What the integrated file system is

The **integrated file system** is a part of OS/400 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 in your server.

The key features of the integrated file system are the following:

- 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. Specifying the path through the directories to an object accesses the object.
- A common interface that allows users and applications to access not only the stream files but also database files, documents, and other objects that are stored in your server.
- A common view of stream files that are stored locally on your server, 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 iSeries server, or a Network File System server.
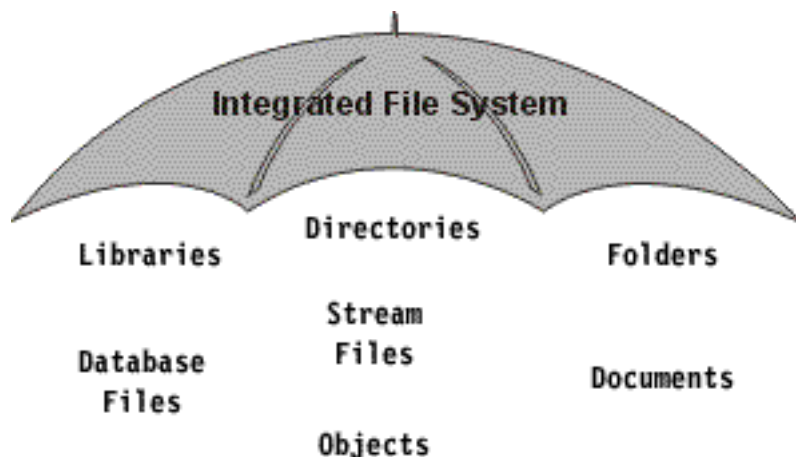
Figure 1. A structure over all information stored in the iSeries server

Why use the integrated file system

The integrated file system enhances the already extensive data management capabilities of OS/400 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:

- Provide fast access to OS/400 data, especially for applications such as Client Access that use the OS/400 file server.
- Allow more efficient handling of types of stream data, such as images, audio, and video.

1

- Provide a file system base and a directory base for supporting UNIX-based open system standards, such as Portable Operating System Interface for Computer Environments (POSIX) and 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-based stream files, and file serving) to be handled as separate file systems, while allowing them all to be managed through a common interface.
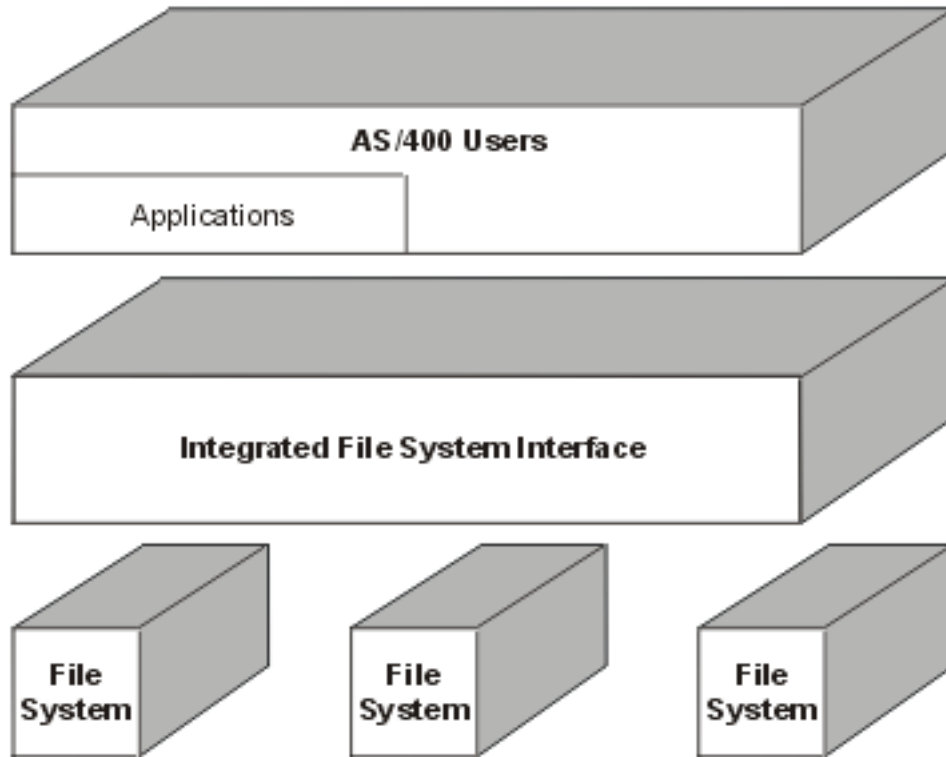
Figure 2. A common interface to separate file systems

- 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 iSeries server 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.

Chapter 2. Integrated file system concepts

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 server'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 iSeries 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.

Stream File

Record-oriented Database File

Record 1

Record 2

Record n

Field 1   Field 2   Field 3   Field n
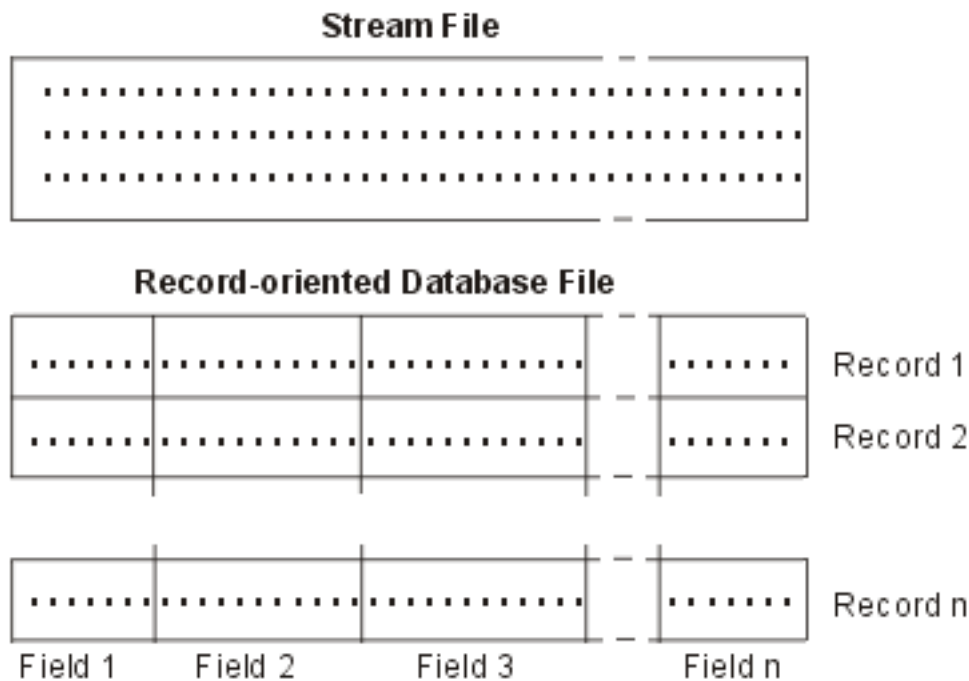
Figure 3. Comparison of a stream file and a record-oriented file

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 server. 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.

For more information on stream files in the integrated file system, see:
- "Copy data between stream files and database files" on page 43.
- "*TYPE1 and *TYPE2 stream files" on page 4.

*TYPE1 and *TYPE2 stream files

A file has one of two format options: *TYPE1 stream file or *TYPE2 stream file.

A *TYPE1 stream file has the same format as stream files created on releases prior to version 4, release 4 of OS/400. It is saved faster than a *TYPE2 stream file to releases prior to version 4 release 4 of OS/400. It has a minimum size of 4096 bytes.

A *TYPE2 stream file has high performance file access and was new in version 4 release 4 of OS/400. It is saved slower than a *TYPE1 stream file to releases prior to version 4 release 4 of OS/400. It has a minimum object size of 4096 bytes. All files created with V4R4 and newer systems are *TYPE2 stream files.

Though *TYPE2 stream files work only with V4R4 and newer systems, you can save a *TYPE2 stream file for restoration on a pre-V4R4 system. However, this process may be slow.

File systems in the integrated file system

A **file system** provides you the support to access specific segments of storage that are organized as logical units. These logical units on your server 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 OS/400 support for accessing database files and various other object types through libraries can be thought of as a file system. Similarly, the OS/400 support for accessing documents (which are really stream files) through the folders structure may 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.

To see a comparison of the features and limitations of each file system, see "File system comparison" on page 57.

The file systems in the integrated file system are:

"root" (/)
> The ″**root**″ **(/)** file system. This file system takes full advantage of the stream file support and hierarchical directory structure of the integrated file system. The root file system has the characteristics of the Disk Operating System (DOS) and OS/2 file systems.

QOpenSys
> The open systems file system. This file system is compatible with UNIX-based open system standards, such as POSIX and 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 supports case-sensitive object names.

UDFS The user-defined file system. This file system resides on the auxiliary storage pool (ASP) or independent auxiliary storage pool (ASP) of your choice. You create and manage this file system.

QSYS.LIB
> The library file system. This file system supports your server's library structure. This file system provides access to database files and all of the other iSeries server object types that the library support manages in the system and basic user ASPs.

Independent ASP QSYS.LIB
> The Independent ASP QSYS.LIB file system. This file system supports your server library structure

in any independent auxiliary storage pools (ASPs) you create and define. This file system provides access to database files and all of the other iSeries server object types that the library support manages.

QDLS The document library services file system. This file system provides access to documents and folders.

QOPT

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

QNetWare

The QNetWare file system. This file system provides access to local or remote data and objects that are stored on a server that runs Novell NetWare 4.10 or 4.11 or to standalone PC Servers running Novell NetWare 3.12, 4.10 4.11 or 5.0. You can dynamically mount NetWare file systems over existing local file systems.

QNTC Windows NT Server file system. This file system provides access to data and objects that are stored on a server running Windows NT 4.0 or higher. It allows iSeries server applications to use the same data as Windows NT clients. This includes access to the data on a Windows NT Server that is running on an Integrated PC Server. See OS/400-AS/400 Integration with Windows NT Server, SC41-5439-01 (SC41-5439) for details.

QFileSvr.400

This file system provides access to other file systems that reside on remote iSeries servers.

NFS Network File System. This file system provides you 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.

You can interact with any of the file systems through a common interface. This interface is optimized for input/output of stream data, in contrast to the record input/output that is provided through the data management interfaces. The provided commands, menus and displays, and application program interfaces (APIs) allow interaction with the file systems through this common interface.
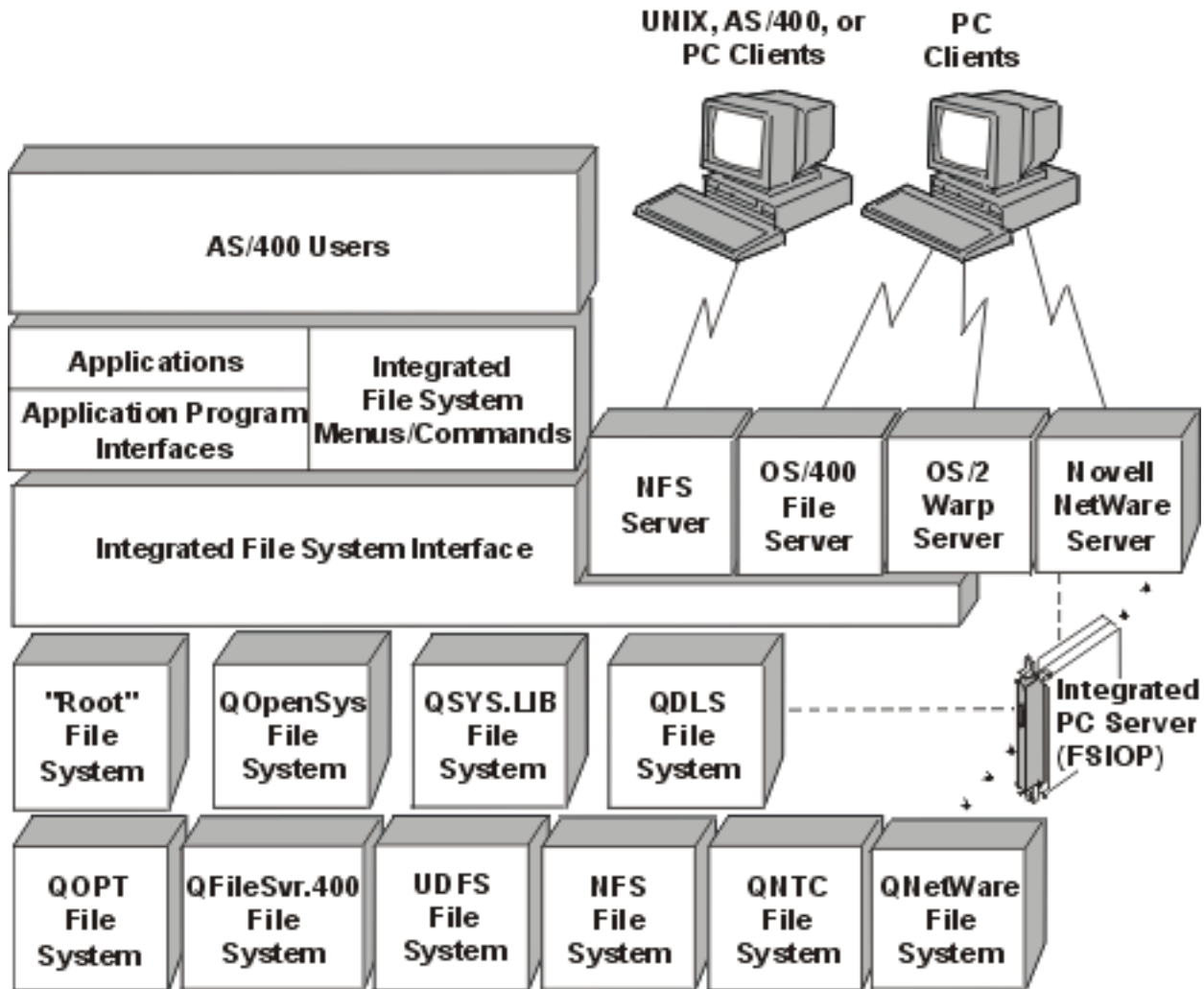
Figure 4. File systems, file servers, and the integrated file system interface

For more information, refer to the following topics and publications:

- Optical Support

- OS/400 Network File System 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 may include other directories.

The integrated file system provides a hierarchical directory structure that allows you to access all objects in your server. 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 sub-directories. Attached to the various directory and sub-directory branches are objects such as files. Locating an object requires specifying a path through the directories to the sub-directory 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 (sub-directories) and all of the objects that are attached to those branches, is referred to as a **sub-tree**. Each file system is a major sub-tree in the integrated file system directory structure. In the QSYS.LIB and Independent ASP QSYS.LIB file systems' sub-trees, a library is handled the same way as a sub-directory. Objects in a library are handled like objects in a sub-directory. Because database files contain objects (database file members), they are handled like sub-directories rather than objects. In the document library services file system (QDLS sub-tree), folders are handled like sub-directories and documents in folders are handled like objects in a sub-directory.

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

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.

Refer to the following topics for more information about integrated file system directories:
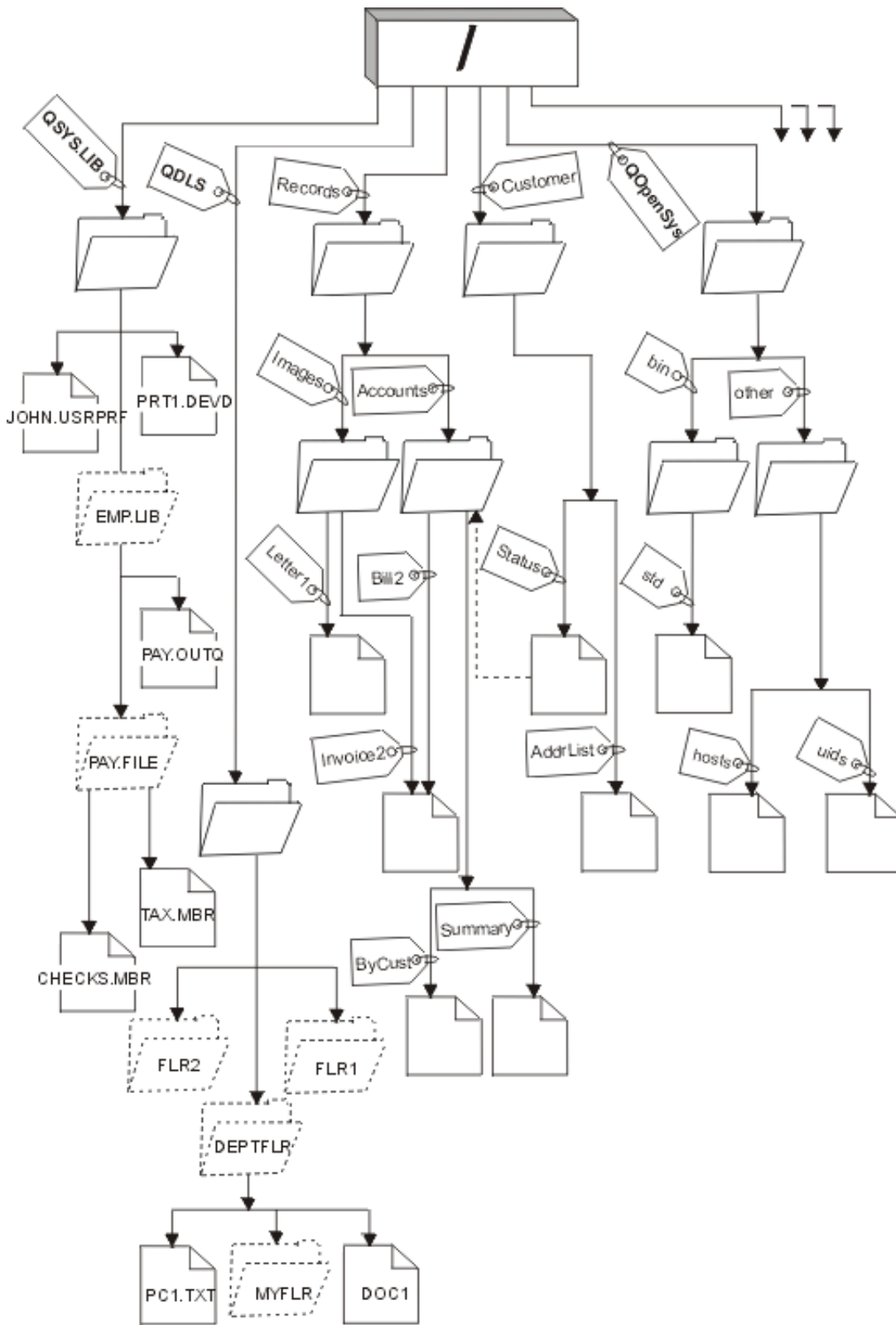
Figure 5. File systems and objects are branches on the integrated file system directory tree

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

The **home directory** is used as the current directory when you sign on 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 would also create the user's home directory. Creating individual home directories for each user under the /home directory is recommended. The /home directory is a sub-directory 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 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.

See the Application programming interfaces (APIs) topic for details on the QWTCHGJB API.

*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. *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
* Less auxiliary storage space (in many cases).

*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:

1. *TYPE2 directories support renaming the case of a name in a monocase file system (for example, renaming from A to a).

2. 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 1 million sub-directories.

3. Using iSeries Navigator, the list of entries are automatically sorted in binary order when you open a directory that has the *TYPE2 format.

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.

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

- A user-defined file system (UDFS) in an independent auxiliary storage pool (ASP) is converted to *TYPE2 format the first time the independent ASP is varied on to a system installed with OS/400 V5R2.
- All other supported file systems except UDFSs on independent ASPs must be converted to *TYPE2 by using the Convert Directory (CVTDIR) command.
- New iSeries servers that are pre-loaded with OS/400 V5R2 have *TYPE2 directories. No conversion is needed for ″root″ (/), QOpenSys, and UDFSs in ASPs 1-32.
- A scratch install of OS/400 V5R2 on an iSeries server has *TYPE2 directories. No conversion is needed for ″root″ (/), QOpenSys, and UDFSs in ASPs 1-32.

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

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

Note: *TYPE2 directories are supported on OS/400 V5R1, but there are some differences from normal *TYPE2 directory support. For more information, see Using *TYPE2 directories in OS/400 V5R1.

For more information about *TYPE2 directories, refer to the following topics:

- Convert to a *TYPE2 directory
- ″Root″, QOpenSys, or UDFS unavailability
- Auxiliary storage requirements
- Symbolic link considerations
- Independent auxiliary storage pools (ASPs)
- Save/restore considerations
- Prepare for *TYPE2 conversion
- Conversion processing
- Example: Convert all file systems (small number of objects)
- Example: Convert all file systems (large number of objects)
- Example: Convert only certain ASPs

Using *TYPE2 directories in OS/400 V5R1

The ″root″ (/), QOpenSys, and user-defined file systems (UDFS) in the integrated file system support the *TYPE2 directory format in 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.

If you have V5R1, you can convert your V5R1 directories to the *TYPE2 directory format. It is recommended that you convert to the *TYPE2 directory format **before** installing a new release of OS/400. This is necessary because the directory conversion may automatically be done during the installation. The impact of an automatic conversion during install is that the time required for the installation will be **significantly** increased.

Note: If you upgrade to OS/400 V5R1 or V5R2, automatic conversion to the *TYPE2 directory format does **not** take place. You do not need to convert your directories prior to these installs.

The support for *TYPE2 directories in V5R1 is available through fixes (PTFs). The conversion utility is slightly different from the 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:

1. Download the informational APAR to your iSeries server and view it. Use the following commands:

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

2. Go to http://www-912.ibm.com Web site to view the informational APAR online. Select **Authorized Program Analysis Reports (APARs) —> V5R1 APARs —> APAR number II13161**.

Convert to a *TYPE2 directory

The CVTDIR command performs the conversion from a *TYPE1 directory to a *TYPE2 directory. Additionally, it provides information on how to convert file systems to the *TYPE2 directory format. CVTDIR does the following:

- Lists the current directory format for existing file systems that support *TYPE2 directories.
- Estimates the time it will take to do the conversion.
- Estimates auxiliary storage requirements for the conversion.
- Converts the file systems to *TYPE2 format. Any existing directories are converted to *TYPE2, and any new directories created after the conversion are *TYPE2.

There are several ways that directories in one of the file systems are converted:

- Manually, by using the CVTDIR command
- Automatically, the first time an independent ASP is varied on to a system that has OS/400 V5R2 installed
- During an IPL, if the system determines that the conversion of a file system was in progress when the system abnormally ended
- During Reclaim Storage (RCLSTG SELECT(*ALL)), if lost *TYPE1 directories are found that are part of a file system that was converted to the *TYPE2 format

″Root″, QOpenSys, or UDFS unavailability

The conversion of the ″root″ (/) or QOpenSys file systems must be done when the system is in a restricted state. When converting UDFSs, the system is not required to be in a restricted state; however, the UDFSs in that ASP are not available during the conversion. The length of time required to perform a conversion is dependent on the size of the file system. Therefore, planning is necessary in order to schedule the best time to perform the conversion. The *ESTIMATE option of the CVTDIR command estimates the length of time needed to convert the specified file system. The time length estimated is the highest estimated value. It estimates the time length based on a conversion run in a job with a single thread. The actual conversion uses multiple threads and should take less time than the estimated time. Typically, file systems that have

more than 40,000 links can be converted in 30 percent to 50 percent of the estimated time. However, the actual time is dependent on the hardware and configuration of the server.

If the system abnormally ends while the CVTDIR command is running, then during the subsequent IPL, the conversion function runs in the SCPF job. The SCPF job does not allow multiple threads to be active. Therefore, when the conversion of a file system must be completed during the IPL, it runs using a single thread. The conversion function runs when SRC C900 2A85 is displayed during the IPL, and status message CPIA089 is displayed, indicating the progress of the conversion.

The conversion function runs during RCLSTG if there are lost *TYPE1 directories from a file system already converted to *TYPE2. The conversion function runs in the job that issues the RCLSTG command. If any lost directories are found that need to be converted, the conversion runs in a single thread because of system restrictions.

Auxiliary storage requirements

Auxiliary storage requirements should be considered before converting the directories in a file system to the *TYPE2 format. There are several issues 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 a number of directories have both a *TYPE1 version and a *TYPE2 version in existence simultaneously. This number is dependent on the iSeries server configuration and the directory structure of the file system being converted.

The *ESTIMATE option on the CVTDIR command will provide information indicating the amount of auxiliary storage estimated to be needed during conversion.

Symbolic link considerations

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 could 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. See Objects renamed for details on object renaming.

Independent auxiliary storage pools (ASPs)

The first time an independent ASP is varied on to a system installed with OS/400 V5R2, the directories are converted to *TYPE2. 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 V5R2 server, run the following API on your V5R1 system when the independent ASP (named ASP_NAME) is varied on and active:

CALL QP0FCVT2 (*ESTIMATE ASP_NAME)

Save/restore considerations

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.

Prepare for *TYPE2 conversion

There are several CL commands and parameters that are recommended before you convert to *TYPE2 directories:

- Backup your file systems

  Follow your normal backup procedures to ensure the data in your file systems are saved before using the *CONVERT option of the CVTDIR command. When saving the root file system, omit the QSYS.LIB, QDLS, QNetWare and QNTC file systems since those file systems will not be converted. A full system save of your iSeries server is not necessary, but if performed will provide the recommended backup of your file systems.

  For more information, see Backup and Recovery .

  This backup needs to be performed only once before the *CONVERT option of the Convert Directory API is used.

- The *ESTIMATE option of the CVTDIR command

  Use the *ESTIMATE option of the CVTDIR command to determine the time required to convert your directories.

  In addition to providing time and auxiliary storage estimates, the *ESTIMATE option has additional benefits. It builds some secondary objects associated with *TYPE1 directories, which allows the conversion to run faster (since it does not need to create them). These secondary objects remain in existence until the *CONVERT option is used to convert the file system to the *TYPE2 format. The *ESTIMATE option also reads through all the directories in the file system, which implicitly verifies the directories. The *ESTIMATE option does not guarantee to find all possible errors that could occur during the actual conversion, but it helps.

  After you run the *ESTIMATE option, check for errors in the job log, and perform any recommended recovery actions before you convert the file system. Running the estimate again after performing the recommended recovery actions is not required, but it may be desired in order to verify that no other problems are found.

- Auxiliary storage considerations

  Check the available auxiliary storage for the ASP that contains the file system being converted. The CVTDIR *ESTIMATE option displays message CPIA090, which indicates how much auxiliary storage is available for the ASP. Additionally, it displays the amount of auxiliary storage that is expected to be needed during conversion. Message CPIA091 is also displayed, which indicates whether or not the total size of *TYPE2 directories in the file system after converting are estimated to be larger or smaller than the existing *TYPE1 directories. The available storage in the ASP should be the sum of the unused auxiliary storage (displayed in message CPIA090), and the difference between *TYPE1 directory size and *TYPE2 directory size (displayed in message CPIA091).

  Alternatively, the available auxiliary storage can be found using the Start System Service Tools (STRSST) command and selecting the Work With Disk Units option.

  Note: If there is only one ASP defined on the system, the Work with System Status (WRKSYSSTS) command is sufficient to show available auxiliary storage information.

  It is a good idea to perform general system cleanup before using any of the options on the CVTDIR command. If there are any directories or files that are no longer needed, remove them before using any options of the CVTDIR command. Doing so frees up auxiliary storage space, provides a more accurate estimate of available auxiliary storage space, and allows the conversion to complete in a shorter amount of time because there are fewer objects that need to be processed.

- Consider changing the job message queue full action to *PRTWRAP for the job that is issuing the CVTDIR command. Doing this does the following:
  1. Prevents the job from abnormally ending in the event the job log fills up
  2. Prints any overlaid messages to a spool file if the job log does wrap; therefore, no important messages are lost

- On systems with independent ASPs: Prior to varying on the independent ASP to a system running OS/400 V5R2, use the V5R1 *ESTIMATE function on all independent ASPs. This provides a time estimate of how long the first vary on of the independent ASP will take after the install. See Independent auxiliary storage pools (ASPs) for more information.

Conversion processing

The CVTDIR command converts *TYPE1 directories to *TYPE2 directories. There are several things to consider during the conversion process:

- Convert ″root″ (/) or QOpenSys
- Convert user-defined file systems
- Create user profiles
- Objects renamed
- User profile considerations

Convert ″root″ or QOpenSys

When converting the ″root″ or QOpenSys file systems, the system **must** be in a restricted state. You cannot use either file system during conversion. All UDFS and NFS file systems are unmounted by the CVTDIR command and are not remounted when the conversion is complete. The Mount command (MOUNT) can be used to remount a UDFS or NFS file system.

Convert user-defined file systems

When the UDFSs in ASPs 1-32 are converted, they are not available to anyone using the system. For each of these ASPs, there is a QASPxx directory in the /dev directory. While the CVTDIR command is running, it removes the QASPxx directory from the namespace to prevent any user from accessing the UDFSs in the ASP. When the CVTDIR command finishes processing all the objects, including the QASPxx directory, the objects are put back in the namespace and made available to users on the system. The UDFSs for the ASP are unmounted by the CVTDIR command and are not remounted when the conversion is complete. The Mount command (MOUNT) can be used to remount a UDFS.

Note: The QASP01 directory exists on every system.

Create user profiles

The conversion function creates user profiles that are used while the conversion function is running. These user profiles have the name QP0FCVxxxx, where xxxx is a number, such as 0001. They are used by the conversion function to own directories in the file system that is converted if the original owner is unable to own their directories.

These user profiles are deleted when the conversion has completed, if possible. Message CPIA08B is sent if ownership of a directory is given to one of these user profiles.

Objects renamed

*TYPE2 directories require the link names to be valid UTF-16 names. This differs from *TYPE1 directories, which have UCS2 Level 1 names. For this reason, invalid or duplicate names may 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 job log listing the original name and the new name. Combined characters or invalid surrogate character pairs contained in a name may cause an object to be renamed.

For more information on UTF-16, please refer to the Unicode homepage (http://www.unicode.org).

Combined Characters: Some characters may be made up of more than one Unicode character. For example, there are characters that have an accent or an umlaut. These characters 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.

Change maximum storage for a user profile: During the directory conversion processing, there are a number of directories that momentarily exist in both formats at the same time and are owned by the same user profile. If the *TYPE2 directory cannot be created because the maximum storage limit for the user profile has been reached, the limit for the user profile is increased. Message CPIA08C is sent to the job log, and conversion continues.

Change 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 one of the alternate user profiles described in Create user profiles. Message CPIA08B is sent to the job log, and conversion continues.

Example: Convert all file systems (small number of objects)

System A has five auxiliary storage pools (ASPs) configured: 1 (the system ASP), 3, 5, 11, and 25. None of the file systems have been converted from *TYPE1 directories to *TYPE2 directories on the system. You would like to convert all of the file systems. The file systems do not contain a large number of objects, so you plan to perform all the steps in a single day.

To convert directories in all file systems that have a small number of objects:

1. Put the system in restricted state.
2. Unmount all user-defined file systems and save all of the file systems using the Save Object (SAV) command.

   ```
   UNMOUNT TYPE(*ALL) MNTOVRDIR(*ALL)
   SAV OBJ(('/*') ('/QSYS.LIB' *OMIT) ('/QDLS *OMIT) ('/QNetWare' *OMIT) ('/QNTC' *OMIT))
   ```
3. Type `CVTDIR OPTION(*ESTIMATE) FILESYS(*ALL) FORMAT(*TYPE2)` on the command line.
4. Check for any error messages from the *ESTIMATE function.
5. Verify all ASPs have enough available auxiliary storage space.
6. Type `CVTDIR OPTION(*CONVERT) FILESYS(*ALL) FORMAT(*TYPE2)` on the command line.

   Note: When converting all file systems (*ALL), message CPAA084 is displayed and asks you to verify that the listed file systems are to be converted.
7. Check for any error messages from the *CONVERT function.
8. Remount all file systems that were unmounted above.
9. Take the system out of restricted state.

Example: Convert all file systems (large number of objects)

System B also has five auxiliary storage pools (ASPs) configured: 1 (the system ASP), 3, 5, 11, and 25. None of the file systems have been converted from *TYPE1 directories to *TYPE2 directories on the system. You would like to convert all of the file systems. The file systems contain a large number of objects, so you plan to perform the conversion steps on two different weekends.

Weekend one:

1. Put the system in restricted state.
2. Unmount all user-defined file systems and save all of the file systems using the Save Object (SAV) command.

   ```
   UNMOUNT TYPE(*ALL) MNTOVRDIR(*ALL)
   ```

   ```
   SAV (OBJ(('/*') ('/QSYS.LIB' *OMIT) ('/QDLS' *OMIT) ('/QNetWare' *OMIT) ('/QNTC' *OMIT))
   ```

   Remount all file systems that were unmounted above.
3. Take the system out of restricted state.

During the week:

4. Type `CVTDIR OPTION(*ESTIMATE) FILESYS(*ALL) FORMAT(*TYPE2)` on the command line.
5. Check for any error messages from the *ESTIMATE function.
6. Verify all ASPs have enough available auxiliary storage space.

Weekend two:

7. Put the system in restricted state
8. Type `CVTDIR OPTION(*CONVERT) FILESYS(*ALL) FORMAT(*TYPE2)` on the command line.

   Note: When converting all file systems (*ALL), message CPAA084 is displayed and asks you to verify that the listed file systems are to be converted.
9. Check for any error messages from the *CONVERT function.
10. Take the system out of restricted state.

Example: Convert only certain ASPs

System C has six auxiliary storage pools (ASPs) configured: 1 (the system ASP), 2, 4, 8, 10, and 30. None of the file systems have been converted on the system. You want to convert the UDFSs in ASPs 4, 10, and 30 only.

To convert the directories in the UDFSs on certain ASPs:

1. Verify the directory format of the file systems. Type `CVTDIR OPTION(*CHECK)` on the command line to do this.
2. Unmount and save the UDFSs being converted using the Save Object (SAV) command. For example:

   ```
   SAV OBJ(('/dev/QASP04/*'))
   ```

   ```
   SAV OBJ(('/dev/QASP10/*'))
   ```

   ```
   SAV OBJ(('/dev/QASP30/*'))
   ```
3. Type `CVTDIR OPTION(*ESTIMATE)  FILESYS(*UDFS) ASP(10) FORMAT(*TYPE2)` on the command line.
4. Check for any error messages from the *ESTIMATE function.
5. Type `CVTDIR OPTION(*ESTIMATE)  FILESYS(*UDFS) ASP(30) FORMAT(*TYPE2)` on the command line.
6. Check for any error messages from the *ESTIMATE function.
7. Verify all ASPs have enough available auxiliary storage space.
8. Type `CVTDIR OPTION(*CONVERT)  FILESYS(*UDFS) ASP(4) FORMAT(*TYPE2)` on the command line.
9. Check for any error messages from the *CONVERT function
10. Remount all file systems in ASP 4 that were unmounted above.

| 11. Type `CVTDIR OPTION(*CONVERT)  FILESYS(*UDFS) ASP(10) FORMAT(*TYPE2)` on the command line.
| 12. Check for any error messages from the *CONVERT function
| 13. Remount all file systems in ASP 10 that were unmounted above.
| 14. Type `CVTDIR OPTION(*CONVERT)  FILESYS(*UDFS) ASP(30) FORMAT(*TYPE2)` on the command line.
| 15. Check for any error messages from the *CONVERT function
| 16. Remount all file systems in ASP 30 that were unmounted above.

Path name

A **path name** (also called a **pathname** on some systems) tells the server 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 back slash (\) 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.
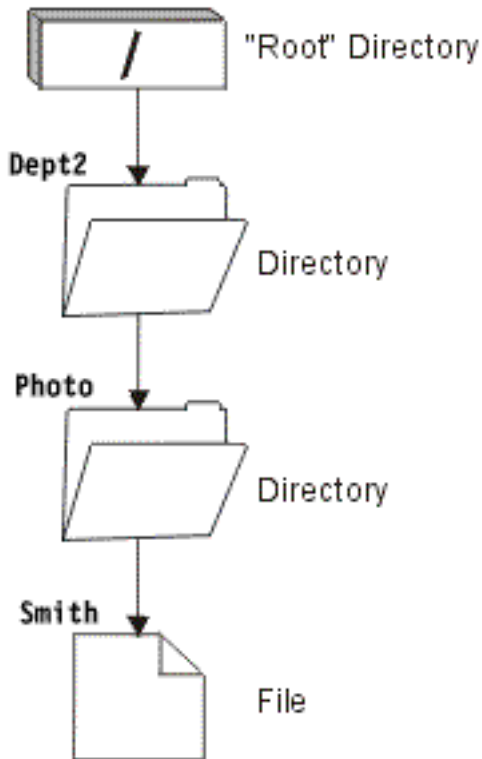
Figure 6. The components of a path name

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 sub-directory 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.

Link

A **link** is a named connection between a directory and an object. A user or a program can tell the server 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 server. 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 only 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.

Once 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 "Current directory and home directory" on page 9). Certain types of links can cross file systems, and can exist without an object existing.

There are two types of links: **hard links** and **symbolic links**.

Refer to the following topics for more information about links:
- Hard link
- Symbolic link
- Comparison: Hard link and symbolic link

Hard link

A **hard link**, which is sometimes called just 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.
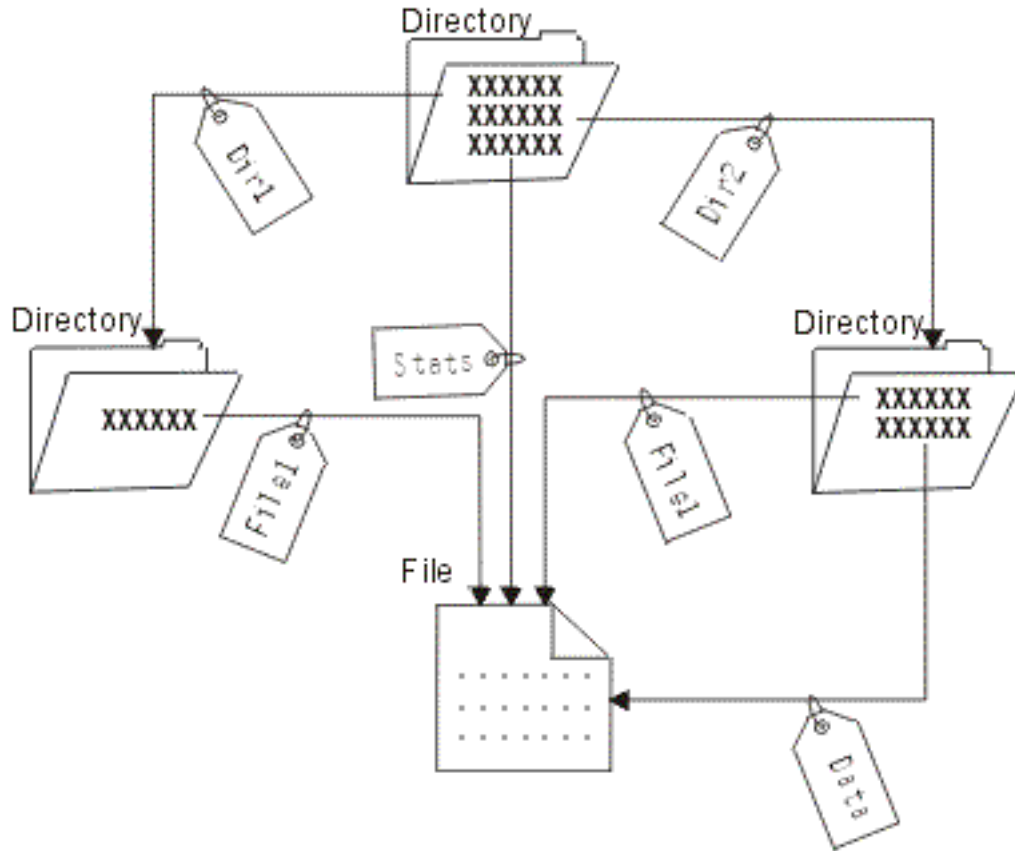
Figure 7. A directory entry defines each hard link.

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 server, 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 server. 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:
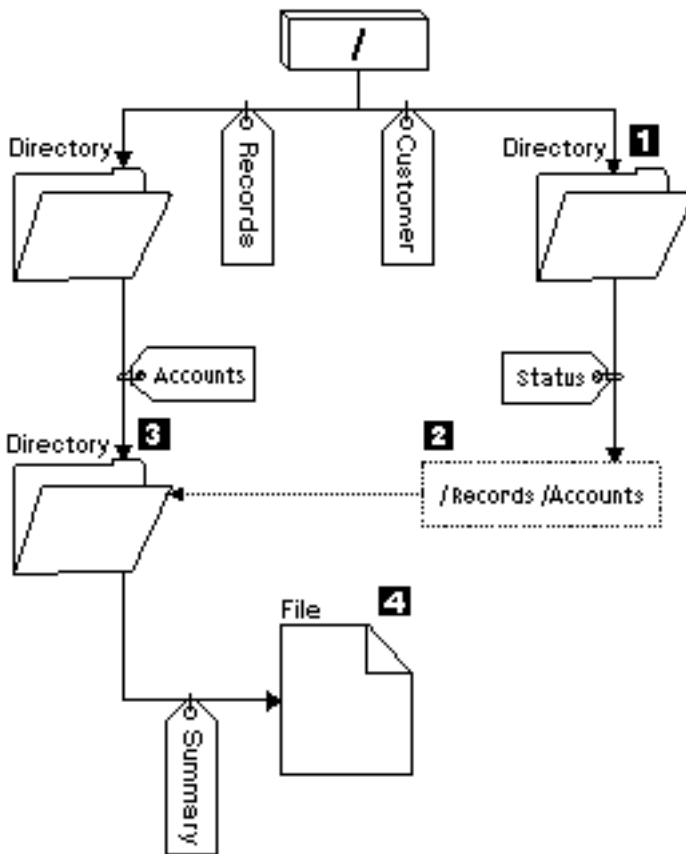
Figure 8. An example of using a symbolic link

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 could 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 could 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 could 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.

See also "Comparison: Hard link and symbolic link" on page 21.

Comparison: Hard link and symbolic link

When using path names in programs, you have a choice of using a hard link or a symbolic link (see "Link" on page 18). 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.
Dynamic objects (attributes change)	Slower. Many of the attributes of an object are stored in each hard link. Changes to a dynamic object, therefore, are slower as the number of hard links to the object increases.	Faster. Changes to a dynamic object are not affected by symbolic links.
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.

Extended attributes

An extended attribute (EA) is information associated with an object that provides additional details about the object. The EA 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 EAs for an object exist only as long as the object exists.

EAs come in many varieties and can be used to contain a variety of information. You may need to be aware of the following three EAs, 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 EA associated with the object.

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

Various objects in the various file systems may or may not have EAs. The QSYS.LIB and Independent ASP QSYS.LIB file systems support three predefined EAs: .SUBJECT, .TYPE, and .CODEPAGE. In the document library services (QDLS) file system, folders and documents can have any kind of EA. Some

folders and documents may have EAs, and some may not. In the "root" (/), open systems (QOpenSys), and user-defined file systems, all directories, stream files, and symbolic links can have EAs of any kind. Some, however, may not have any EAs at all.

The Work with Object Links (WRKLNK) command can be used to display the .SUBJECT extended attribute (EA) for an object. There is no other integrated file system support through which applications or users can access and change EAs. 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.

EAs associated with some objects in the QDLS can, however, be changed through interfaces provided by the hierarchical file system (HFS). See "Document Library Services File System (QDLS)" on page 72 and "Optical File System (QOPT)" on page 73 for more information about these file systems.

If a client PC is connected to an iSeries server 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 EAs of any file object. OS/2 users can also change the EAs of an object on the desktop by using the settings notebook; that is, by selecting `Settings` on the pop-up menu associated with the object.

If you define extended attributes, use the following naming guidelines:
- The name of an EA can be up to 255 characters long.
- Do not use a period (.) as the first character of the name. An EA whose name begins with a period is interpreted as a standard system EA.
- To minimize the possibility of name conflicts, use a consistent naming structure for EAs. The following form is recommended:

      `CompanyNameProductName.Attribute_Name`

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 iSeries servers and connected devices that have different character encoding schemes (code pages). Your server stores the characters in the names in a 16-bit form that is known as UCS2 Level 1 (that is also called **Unicode**) for *TYPE1 directories and UTF-16 for *TYPE2 directories. Refer to *TYPE2 directories for more information about the directory formats. 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 proper character representation in the code page being used. The names of extended attributes associated with each object are also handled the same way.
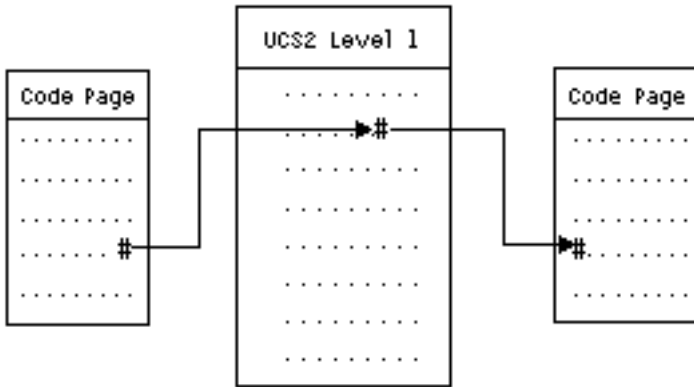
Figure 9. Keeping characters the same across encoding schemes

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

Chapter 3. Access the integrated file system using the traditional system interface

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 server by using menus and displays or by using control language (CL) commands. Additionally, you can use application program 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 used for managing and administering your server from your Windows desktop.

There are several ways to interact with the integrated file system:

Using APIs
The application program interfaces (APIs) that perform operations on integrated file system directories and stream files are in the form of C language functions.

Using CL commands
CL commands can operate on files and other objects in any file system that are accessible through the integrated file system.

Using iSeries 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 server.

Using iSeries Navigator
iSeries Navigator is the graphical user interface for managing and administering your servers from your Windows desktop.

Using a PC
If your PC is connected to an iSeries server, you can interact with the directories and objects of the integrated file system as if they were stored on your PC.

Perform operations using iSeries 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 server. To display integrated file system menus:

1. Sign on to your server.
2. Press **Enter** to continue.
3. Select the **Files, Libraries, and Folders** option from the iSeries main menu.
4. Select the **Integrated File System** option from the Files, Libraries, and Folders menu.

From here, you can work with Directory commands, Object commands, or Security commands in the integrated file system, depending upon 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 server by performing the following steps:

1. Type `GO DATA` on any command line to display the Files, Libraries, and Folders menu.
2. Select the option `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 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
- 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. For restrictions in particular file systems, see "File systems in the integrated file system" on page 4.

Refer to the following topic for more information on integrated file system menus and displays:

- Path name rules for CL commands and displays

Perform operations using CL commands

All of the operations that you can do through the integrated file system menus and displays (see "Perform operations using iSeries menus and displays" on page 25) 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 on 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 system (UDFS)" on page 63 and "Network File System (NFS)" on page 83. 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 2. 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 Journaled Changes. Uses journal entries to apply changes that have occurred since a journaled object was saved or to apply changes up to a specified point.	

Table 2. Integrated File System Commands (continued)

Command	Description	Alias
CHGATR	Change Attribute. Change 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
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 on converting integrated file system directories from *TYPE1 format to *TYPE2 format, or perform a conversion.	
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.	
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.	
EDTF	Edit Stream File. Edits a stream file or a database file.	
ENDJRN[2]	End Journal. End 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.	

Table 2. Integrated File System Commands (continued)

Command	Description	Alias
ENDRPCBIND	End RPC Binder Daemon. Ends the Remote Procedure Call (RPC) RPCBind daemon.	
MOV	Move. Moves an object to a different directory	MOVE
RLSIFSLCK	Release Integrated File System Locks. Releases all NFS byte-range locks held by a 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)	
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. Start 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	
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	

Note:

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 iSeries Information Center for more information.

Refer to the following topics for more information on integrated file system CL commands and restrictions on using these commands in particular file systems:

- File systems in the integrated file system
- Path name rules for CL commands and displays
- The CL topic in the iSeries Information Center

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. Following 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 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 apostrophe (') 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 (/) or back slash (\); for example:

      'Dir1/Dir2/Dir3/UsrFile'

  or

      'Dir1\Dir2\Dir3\UsrFile'

- The / and \ characters and nulls cannot be used in the individual components of the path name (because the / and \ are used as separators). Lowercase letters are not changed to uppercase letters by the commands. The name may or may 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 will accept object names up to 255 characters long and path names up to 5000 characters long.

  See File systems in the integrated file system for path name limits in each file system.
- A / or \ character 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 / or \ character, 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 slash (or backslash) 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 slash (or backslash) 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 iSeries server 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 there are no other characters 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/PAY.LIB/TAX.FILE'
```

See "Library file system (QSYS.LIB)" on page 67 for more details.

- 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
```

See "Independent ASP QSYS.LIB" on page 69 for more details.

- The path name must be enclosed in additional sets of apostrophes (') or quotation marks (") if any of the following characters is used in a component name:
  - Asterisk (*)
  - Question mark (?)
  - Apostrophe (')
  - 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 <u>not recommended</u> because the meaning of the character in a command string could be confused and it is more likely that the command string will 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:

```
crtdir 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 (see "Perform operations using APIs" on page 47).

For restrictions on using a particular command, see the command help or the Control language (CL) topic in the iSeries Information Center.

Perform operations using a PC

If your PC is connected to an iSeries server, 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 server to the PC by selecting the object in the server drive and dragging the object to the PC drive.

Any objects that are copied between an iSeries server and PCs by using the Windows interface may be automatically converted between EBCDIC and ASCII. EBCDIC is extended binary-coded decimal interchange code, and ASCII is the American National Standard Code for Information Interchange. iSeries Access can be configured to automatically perform this conversion, and can even specify the conversion be performed on files with a specific extension. Since OS/400 V4R4, iSeries NetServer can also be configured to perform conversion on files.

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

If you are connected to an iSeries server using your PC, the integrated file system makes your server'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). Your PC uses Windows file sharing clients to access iSeries NetServer, which runs on your iSeries server.

Transfer files using FTP
The FTP client allows you to transfer files that are found on your iSeries server, including those in the ″root″ (/), QSYS.LIB, Independent ASP QSYS.LIB, QOpenSys, QOPT, and QFileSvr.400 file systems. It also allows you to transfer folders and documents in the document library services (QDLS) file system.

Work with files using iSeries Navigator
iSeries Access includes iSeries Navigator, which connects to an iSeries server and makes the integrated file system available to the PC. iSeries Navigator is the graphical user interface for managing and administering your iSeries server from your Windows desktop.

Work with files using iSeries NetServer
iSeries NetServer is a part of OS/400 that allows the file and print sharing built into Windows clients to work with your server.

Note: The new version of iSeries Access relies entirely on NetServer to access the integrated file system. NetServer support is available only for TCP/IP connections to an iSeries server that run OS/400 V4R2 and above.

Transfer files using FTP

The File Transfer Protocol (FTP) client allows you to transfer files that are found on your iSeries server, including those in the ″root″, QOpenSys, QSYS.LIB, Independent ASP QSYS.LIB, QOPT, and QFileSvr.400 file systems. It also allows you to transfer folders and documents in the document library services (QDLS) file system. The FTP client may 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 server.

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)
- Windows NT Server 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.

For information about FTP, see the following topics in the **Networking** category of the iSeries Information Center:

- FTP
- Transfer files with FTP

Work with files using iSeries NetServer

iSeries Support for Windows Network Neighborhood (iSeries NetServer) is an IBM Operating System/400 Version 5 (OS/400) function that enables Windows clients to access OS/400 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 iSeries. PC clients on a network simply 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 may also seamlessly access data and printers through iSeries NetServer. Samba file systems (smbfs) may be mounted from iSeries NetServer in a similar manner to mounting NFS file systems from the iSeries. For more information, see the iSeries NetServer topic in the iSeries Information Center.

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

Note: When connecting using iSeries NetServer, the server name may be different than the name used by iSeries Access. For example the iSeries NetServer name may be `QAS400X`, and the path to work with files may be `\\QAS400X\QDLS\MYFOLDER.FLR\MYFILE.DOC`. However, the iSeries Access name may be `AS400X`, and the path to work with files may 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 server name. For example, if you share the `/home/fred` directory with the name `fredsdir`, a user would be able to access that directory from the PC with the name `\\QAS400X\FREDSDIR`, or from a LINUX client with the name `//qas400x/fredsdir`.

The ″root″ (/) file system provides much better performance for PC file serving than other iSeries file systems. You may want to move files to the ″root″ (/) file system. See considerations for moving objects to another file system for more information.

For more information about iSeries NetServer and file shares, see the following topics in the **Networking** category of the iSeries Information Center:

- iSeries NetServer
- iSeries NetServer file shares
- Accessing iSeries NetServer file shares with a Windows PC client

Move objects to another file system

Before using the integrated file system to move objects between file systems, review the "Considerations for moving objects to another file system".

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 (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 (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.

Considerations for moving objects to another file system

Each file system has its own unique characteristics. However, moving objects to a different file system may mean losing the advantages of the file system in which the objects are currently stored. You may want to move objects from one file system to another to take advantage of those characteristics. Before

moving objects to another file system, you should be familiar with the file systems on the integrated file system and their characteristics. For more information, see "File systems in the integrated file system" on page 4.

You should also consider the following:

- 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 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. QDLS supports many "office" attributes, but other file systems do not. Therefore, QDLS is a good place to keep your office documents.

  Good candidates for moving are the PC files that are in 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.

Directories provided by the integrated file system

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

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

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

You can use the user displays and commands that support the integrated file system to manage the /tmp directory and its files. For example, you can use the Work with Object Links display or the WRKLNK command to copy, remove, or rename the /tmp directory or files 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 program interfaces (API) that support the integrated file system to manage /tmp and its files (see "Perform operations using APIs" on page 47). For example, the application program can remove a file in /tmp by using the unlink() API.

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

/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. See "Current directory and home directory" on page 9 for more information.

/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 product 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 product 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.

Chapter 4. Access the integrated file system with 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 iSeries server to another iSeries server. 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:

Work with files and folders

Work with file shares

Work with user-defined file systems

Journal objects

Check in a file

To check in a file:
1. In **iSeries Navigator**, right-click the file that you want to check in.
2. Select **Properties**.
3. Select the **File Properties –> Use Page**.
4. Click **Check In**.

Check out a file

To check out a file:

1. In **iSeries Navigator**, right-click the file that you want to check out.
2. Select **Properties**.
3. Select the **File Properties –> Use Page**.
4. Click **Check Out**.

Set up permissions to a file or folder

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.

To set permissions to a file or folder:

1. Expand the system you want to use in the **iSeries Navigator** window .
2. Expand **File Systems**.
3. Expand **Integrated File System**. Continue to expand until the object for which you want to add permissions is visible.
4. Right-click the object for which for you want to add permissions and select **Permissions**.
5. Click **Add** on the **Permissions** dialog.
6. 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.
7. Click **OK**. This will add the users or groups to the top of the list.
8. Click on the **Details** button to implement detailed permissions.
9. Apply the desired permissions for the user by checking the box by the appropriate check box.
10. Click **OK**.

Set 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. The integrated file system can convert a data file when it is transferred between an iSeries 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:

1. Expand the system you want to use in **iSeries Navigator**.
2. Expand **File Systems**.
3. Right-click **Integrated File System** and select **Properties**.
4. Enter the file extension that you want to convert automatically in the **File extensions for automatic text file conversion** text box and click **Add**.
5. Repeat step 4 for all file extensions that you want to convert automatically.
6. Click **OK**.

Send a file or folder to another system

To send a file or folder to another system:

1. Expand the system you want to use in **iSeries Navigator**.
2. Expand **File Systems**.
3. Expand **Integrated File System**. Continue to expand until the file or folder you want to send is visible.

4. Right-click on 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.

5. Expand the list of available systems and groups.

6. 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.

7. Click **OK** to send the file or folder with the current default package definitions and schedule information.

You can also "Change options for the package definition" or "Schedule a date and time to send your file or folder".

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.

Change options for the package definition

A package definition allows you to group together a set of OS/400 objects or integrated file system files. 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:

1. Complete the steps for "Send a file or folder to another system" on page 38.

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 topics:

• "Schedule a date and time to send your file or folder".

Schedule a date and time to send your file or folder

Using the scheduler function gives you the flexibility to do your work when it's convenient for you to do it. To schedule a date and time to send your file or folder:

1. Complete the steps for "Send a file or folder to another system" on page 38.

2. Click **Schedule**.

3. Select the options for when you want to send the file or folder.

Create a folder

To create a folder:

1. Expand the system you want to use in **iSeries Navigator**.

2. Expand **File Systems**.

3. Expand **Integrated File System**.

4. Right-click on the file system to which you want to add the new folder and select **New Folder**.

5. Type a new name for the object in the **New Folder** dialog.

6. Click **OK**.

When you create a folder on the iSeries server, you need to consider whether or not you want to protect the new folder (or object) with journal management. See the Journal management topic for more information.

Related topics:
- Start journaling
- End journaling

Remove a folder

To remove a folder:

1. Expand the system you want to use in **iSeries Navigator**.
2. Expand **File Systems**.
3. Expand **Integrated File System**. Continue to expand until the file or folder you want to remove is visible.
4. Right-click on the file or folder and select **Delete**.

Create a file share

A file share is a directory path that iSeries NetServer shares with PC clients on the iSeries network. A file share can consist of any integrated file system directory on iSeries.

To create a file share:

1. Expand your system in **iSeries Navigator**.
2. Expand **File Systems**.
3. Expand **Integrated File System**.
4. Expand the file system that contains the folder for which you want to create a share.
5. Right-click the folder for which you want to create a share and select **Sharing**.
6. Select **New Share**.

Change a file share

A file share is a directory path that iSeries NetServer shares with PC clients on the iSeries network. A file share can consist of any integrated file system directory on iSeries.

To change a file share:

1. Expand your system in **iSeries Navigator**.
2. Expand **File Systems**.
3. Expand **Integrated File System**.
4. Expand the folder that has the share defined for it that you want to change.
5. Right-click the folder that has the share defined for it that you want to **Sharing**.
6. Select **New Share**.

Create 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.

To create a new user-defined file system (UDFS):

1. Expand your system in **iSeries Navigator**.
2. Expand **File Systems**.
3. Expand **Integrated File System**.
4. Expand **Root**.
5. Expand **Dev**.
6. Click the auxiliary storage pool (ASP) that you want to contain the new UDFS.
7. Select **New UDFS** from the **File** menu.
8. Specify the UDFS name, description (optional), auditing values, default file format, and whether the files in the new UDFS will have case-sensitive file names on the **New User-Defined File System** dialog.

Mount a 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. To access or view the data stores in a UDFS, you must mount the UDFS.

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 can not be mounted over.

To mount a user-defined file system (UDFS):

1. Expand your system in **iSeries Navigator**.
2. Expand **File Systems**.
3. Expand **Integrated File System**.
4. Expand **Root**.
5. Expand **Dev**.
6. Click the auxiliary storage pool (ASP) that contains the UDFS that you want to mount.
7. Right-click the UDFS that you want to mount in the **UDFS Name** column of Operations Navigator's right pane.
8. Select **Mount**.

If you like to drag and drop, you can mount a UDFS by dragging it to a folder within the integrated file system on the same server. You cannot drop the UDFS on `/dev`, `/dev/QASPxx`, `/dev/asp_name`, another system, or the desktop.

Unmount 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.

To unmount a user-defined file system (UDFS):

1. Expand your system in **Operations Navigator**.
2. Expand **File Systems**.
3. Expand **Integrated File System**.
4. Expand **Root**.
5. Expand **Dev**.
6. Click the auxiliary storage pool (ASP) that contains the UDFS that you want to unmount.
7. Right-click the UDFS that you want to unmount in the **UDFS Name** column of iSeries Navigator's right pane.
8. Select **Unmount**.

Start journaling

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.

To start journaling on an object:

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**.

For more detailed information on journaling integrated file system objects, refer to Journal management in the iSeries Information Center.

End journaling

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. See Start journaling for more information on how to start journaling on an object. Once journaling has started on an object, for whatever reasons, you may want to end journaling on this object.

To end journaling on an object:

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**.

For more detailed information on journaling integrated file system objects, refer to Journal management in the iSeries Information Center.

Chapter 5. Programming support for the integrated file system

The addition of the integrated file system to the iSeries server does not affect existing iSeries server applications. The programming languages, utilities, and system support such as data description specifications operate the same as they did before the addition of the integrated file system.

To take advantage of the stream files, directories, and other support of the integrated file system, however, you must use a set of C language application program 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, or APIs.

The following topics explain how to use copy functions with integrated file system stream files, and introduce APIs as a way to access integrated file system functions:
- Copy data between stream files and database files
- Copy data between stream files and save files
- Perform operations using APIs
- Socket support
- Naming and international support
- Data conversion

Copy 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 may 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.

There are several ways to copy data between stream files and database files in the integrated file system:
- Copy data using CL commands
- Copy data using APIs
- Copy data using data transfer function

Copy 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
- CPYTOIMPF and CPYFRMIMPF

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 target file exists) to copy between the source physical file and the stream file.

Here is an example:
```
CPYTOIMPF  FROMFILE(DB2FILE) TOFILE(EXPFILE) DTAFMT(*DLM)
        FLDDLM(';') RCDDLM(X'07') STRDLM('"') 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 FLDDLM, RCDDLM and STRDLM 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 server. However, the interfaces are complex.

For more information, see the command help or the Command language (CL) topic in the iSeries Information Center.

Copy 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. See the Integrated File System APIs topic in the iSeries Information Center for more information.

Copy data using data transfer function

The iSeries Access data transfer applications 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 product and requires use of both PC and iSeries server resources and communications between the two.

If you have iSeries Access installed on the PC and your server, 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 based on an existing database file, into an externally-described database file, or into a new database file definition and file.

The following tasks help you copy and transfer data using the data transfer applications:
- Transfer data from a database file to a stream file
- Transfer data from a stream file to a database file
- Transfer data into a newly created database file definition and file
- Create a format description file

Transfer data from a database file to a stream file

To transfer a file from a database file to a stream file on your server:

1. Establish a connection to the server.
2. Map a network drive to the appropriate path in the iSeries file system.
3. Select **Data Transfer from iSeries server** from the iSeries Access for Windows window.
4. Select the server you want to transfer from.
5. Select the file names, using the iSeries database library and file name to copy from, and the network drive for the location of the resulting stream file. You can also choose **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 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.

Transfer data from a stream file to a database file

To transfer data from a stream file to a database file on your server:

1. Establish a connection to the server.
2. Map a network drive to the appropriate path in the iSeries file system.
3. Select **Data Transfer to iSeries server** from the iSeries Access for Windows window.

4. Select the PC file name 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 server 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 server, 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 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 an .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 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.

Transfer data into a newly created database file definition and file

To transfer data into a newly created database file definition and file:

1. Establish a connection to the server.

2. Map a network drive to the appropriate path in the iSeries file system.

3. Select **Data Transfer to iSeries server** from the iSeries Access for Windows window.

4. Open the **Tools** menu of the Data Transfer to iSeries server application.

5. Select **Create iSeries database file**.

   A wizard will appear that allows you to create a new iSeries database file from an existing PC file. You will be required to specify the name of the PC file from which the iSeries file will be based, the name of the iSeries file which 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 server.

Create a format description file

If you are moving data to an existing database file definition on the server, 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.

4. Now, you can use the Data Transfer to iSeries server function. Specify this .FDF file with the source stream file you want to transfer.

Copy 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 to 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 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 could 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).

Perform operations using APIs

The application program interfaces (APIs) that perform operations on integrated file system directories and stream files 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/400:

- Integrated file system C functions that are included in OS/400.
- C functions provided by the ILE C/400 licensed program.

The integrated file system functions operate only through the integrated file system stream I/O support. The following APIs are supported:

Table 3. 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
fchdir()	Change current directory by descriptor
fchmod()	Change file authorizations by descriptor
fchown()	Change owner and group of file by descriptor
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)

Table 3. Integrated File System APIs (continued)

Function	Description
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
getwpnam()	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)
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
mkdir()	Make directory
mkfifo()	Make FIFO special file
open()	Open file
open64()	Open file (large file enabled)
opendir()	Open directory
pathconf()	Get configurable path name variables
pipe()	Create interprocess channel with sockets
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

Table 3. Integrated File System APIs (continued)

Function	Description
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)
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)

Table 3. Integrated File System APIs (continued)

Function	Description
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 *new* if it exists (using NLS-enabled path name)
QlgRenameUnlink()	Rename file or directory, unlink *new* 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
Qp0lCvtPathToSYSObjName()	Resolve integrated file system path name into QSYS Object Name
Qp0lFLOP()	Perform miscellaneous operations on objects
Qp0lGetAttr()	Get system attributes for an object
Qp0lGetPathFromFileID()	Get path name of object from its file ID
Qp0lOpen()	Open file with NLS-enabled path name
Qp0lProcessSubtree()	Process directories or objects within a directory tree
Qp0lRenameKeep()	Rename file or directory, keep *new* if it exists
Qp0lRenameUnlink()	Rename file or directory, unlink *new* if it exists
QP0LROR()	Retrieve object references
Qp0lSaveStgFree()	Save objects data and free its storage
Qp0lSetAttr()	Set system attributes for an object
Qp0lUnlink()	Unlink file with NLS-enabled path name
qsysetegid()	Set effective group ID
qsyseteuid()	Set effective user ID
qsysetgid()	Set group ID
qsysetregid()	Set real and effective group IDs

Table 3. Integrated File System APIs (continued)

Function	Description
qsysetreuid()	Set real and effective user IDs
qsysetuid()	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 OS/400 sockets. For restrictions on use of these functions for particular file systems, see "File systems in the integrated file system" on page 4. For an example program using integrated file system C functions, see Appendix B, "Example program using integrated file system C functions" on page 97.

Refer to the following topics for more information on integrated file system APIs:

- ILE C/400 functions
- Large file support for APIs
- Path name rules for APIs
- File descriptor
- Security
- The Application programming interfaces (APIs) topic in the iSeries Information Center

ILE C/400 functions

ILE C/400 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/400 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 ILE C/400 C functions use the integrated file system functions to perform I/O.
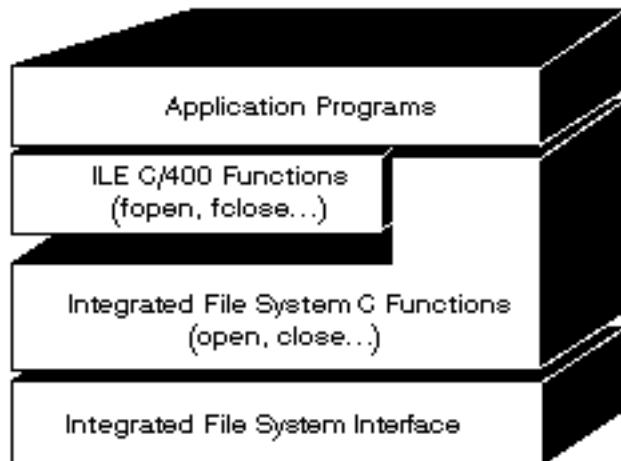
Figure 10. ILE C/400 functions use the integrated file system stream I/O functions.

For more information about using ILE C/400 functions with integrated file system stream I/O, see the

publication WebSphere Development Studio: ILE C/C++ Programmers Guide . For details on each ILE C/400 C function, see the publication WebSphere Development Studio: C/C++ Language Reference

.

Large file support for APIs

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 256 gigabytes in the ″root″ (/), Open Systems File System (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. For details on each 64-bit API, see the Integrated File System APIs topic in the iSeries Information Center.

The following are provided to allow applications to use large file support:
1. 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.

2. 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.

Path name rules for APIs

When using an integrated file system or ILE C/400 API to operate on an object, you identify the object by supplying its directory path. Following 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 back slash (\) 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. See "File system comparison" on page 57 for the limits in 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 iSeries server 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'
    ```

    Note that this only applies to relative path names where there are no other characters 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
    ```

See "Library file system (QSYS.LIB)" on page 67 for more details.
- 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
    ```

See "Independent ASP QSYS.LIB" on page 69 for more details.
- Do not use a colon (:) in path names. It has a special meaning within the server.

| • Unlike path names in integrated file system commands (see "Path name rules for CL commands and
| displays" on page 29), an asterisk (*), a question mark (?), an apostrophe ('), a quotation mark ("), and a
| tilde (~) have no special significance. They are handled as if they are just another character in a name.
| The only APIs that are an exception to this rule are QjoEndJournal() and QjoStartJournal.

File descriptor

When using ILE C/400 stream I/O functions as defined by the American National Standards Institute
(ANSI) to perform operations on a file, you identify the file through 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 *fildes* 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
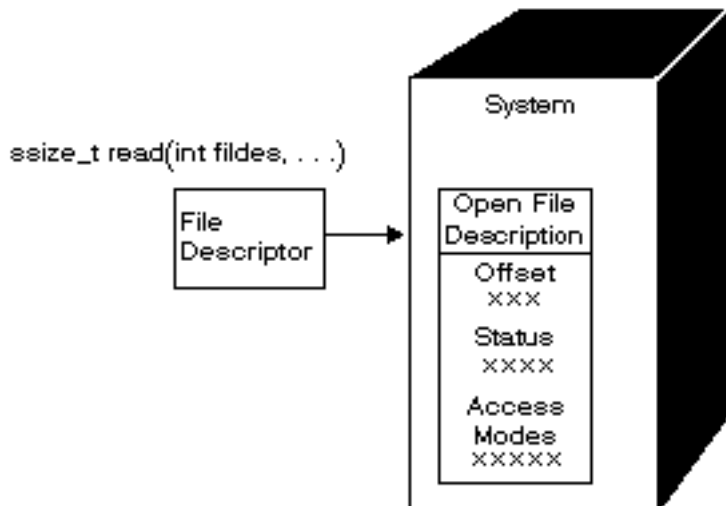than one file descriptor, but a file descriptor can refer to only one open file description.

Figure 11. File descriptor and open file description

If an ILE C/400 stream I/O function is used with the integrated file system, the ILE C/400 run-time 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. For a description of
these functions, see Sockets Programming , or the Sockets APIs topic in the iSeries Information Center.

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 server 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 .

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. These functions and the concepts of sockets in general are described in the Socket Programming topic in the iSeries Information Center.

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.

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 (see "Name continuity" on page 22). When the name is used, it is converted to the proper 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 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.

For more information about code pages, see the Globalization topic in the iSeries Information Center.

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.

Chapter 6. File systems in the integrated file system

The file systems in the integrated file system are:
- "root"
- QOpenSys
- UDFS
- QSYS.LIB
- Independent ASP QSYS.LIB
- QDLS
- QOPT
- QNetWare
- QNTC
- QFileSvr.400
- NFS

For an overview of each file system, see File system comparison.

File system comparison

Table 4 and Table 5 on page 58 summarize the features and limitations of each file system.

Table 4. File System Summary (Part 1 of 2)

Capability	/ ("root")	QOpenSys	QSYS.LIB[16]	QDLS	QNTC
Standard part of OS/400	Yes	Yes	Yes	Yes	Yes
Type of file	Stream	Stream	Record [12]	Stream	Stream
Integrated with OfficeVision (for example, file can be mailed)	No	No	No	Yes	No
Access through OS/400 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	255 char	10.6 char [6]	8.3 char [7]	255 char
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	64KB
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

Table 4. File System Summary (Part 1 of 2) (continued)

Capability	/ ("root")	QOpenSys	QSYS.LIB[16]	QDLS	QNTC
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

Notes:

1. The file server I/O processor is hardware used by LAN Server.

2. When accessed through the OS/400 file server.

3. When accessed through a LAN Server client PC. Access using iSeries APIs is comparatively slow.

4. The QSYS.LIB file system has a maximum path name length of 55 characters. See "Library file system (QSYS.LIB)" on page 67 for details. The Independent ASP QSYS.LIB file system has a maximum path length of 66 characters. See "Independent ASP QSYS.LIB" on page 69 for details.

5. See "Document Library Services File System (QDLS)" on page 72 for details.

6. Up to 10 characters for the object name and up to 6 characters for the object type. See "Library file system (QSYS.LIB)" on page 67 for more details.

7. Up to 8 characters for the name and 1 to 3 characters for the file type extension (if any). See "Document Library Services File System (QDLS)" on page 72 for details.

8. Assuming an absolute path name that begins with / followed by the file system name (such as /QDLS...).

9. 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.

10. In practice, directory levels are limited by program and system space limits.

11. Except a directory, which can have only one link to another directory.

12. The user spaces in QSYS.LIB and Independent ASP QSYS.LIB file systems support stream file input and output.

13. 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.

14. QSYS.LIB and Independent ASP QSYS.LIB file systems may support journaling different object types than the root, UDFS, and QOpenSys file systems. See the Journal management topic in the iSeries Information Center for more information about journaling objects that reside in the QSYS.LIB or Independent ASP QSYS.LIB file systems.

15. *TYPE2 directories have a limit of one million links per object. *TYPE1 directories have a limit of 32, 767 links per object. For more information, see *TYPE2 directories.

16. Data in this column refers to both the QSYS.LIB file system and the Independent ASP QSYS.LIB file system.

Abbreviations

    char = characters

    B = bytes   KB = kilobytes   MB = megabytes   GB = gigabytes

Table 5. File System Summary (Part 2 of 2)

Capability	QOPT	QFileSvr.400	UDFS	NFS	QNetWare
Standard part of OS/400	Yes	Yes	Yes	Yes	No
Type of file	Stream	Stream	Stream	Stream	Stream
Integrated with OfficeVision (for example, file can be mailed)	No	No	No	No	No
Access through OS/400 file server	Yes	Yes	Yes	Yes	Yes

Table 5. File System Summary (Part 2 of 2) (continued)

Capability	QOPT	QFileSvr.400	UDFS	NFS	QNetWare
Direct access through the Integrated PC Server (FSIOP) [1]	No	No	No	No	Yes
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	Varies [2]	255 char [13]
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	No	Yes	No	No
Supports object journaling	No	No	Yes	No	No

Table 5. File System Summary (Part 2 of 2) (continued)

Capability	QOPT	QFileSvr.400	UDFS	NFS	QNetWare
Notes:					

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 OS/400 file server

4. See "Optical File System (QOPT)" on page 73 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 iSeries 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. For more information, see *TYPE2 directories.

Abbreviations

   char = characters

   B = bytes   KB = kilobytes   MB = megabytes   GB = gigabytes

"root" (/) file system

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

In addition, it:
- Is optimized for stream file input/output.
- Supports multiple hard links and symbolic links.
- Supports local sockets.
- Supports *OOPOOL objects.
- Supports threadsafe application program interfaces (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.

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.

For more information on the ″root″ (/) file system, see Use the "root" (/) file system.

Use the "root" (/) file system

The "root" (/) file system can be accessed through the integrated file system interface using either the
OS/400 file server or the integrated file system commands, user displays, and C language 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 server searches for names.

Path names in the ″root″ (/) file system
- Path names have the following form:

      Directory/Directory . . . /Object

- Components of the path names after the sharename follow Windows NT path name rules. The directory
  names and object name combined can fill up to the total path length of 255 characters.

- 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 (see "Name continuity" on page 22). Refer to *TYPE2
| directories for more information about the directory formats.

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.

For a description of links, see "Link" on page 18.

Use integrated file system commands in the ″root″ (/) file system
All of the commands listed in "Perform operations using CL commands" on page 26 and the displays
described in "Perform operations using iSeries menus and displays" on page 25 can operate on the "root"
(/) file system. However, it may not be safe to use these commands in a multi-thread capable process.

Use integrated file system APIs in the ″root″ (/) file system
All of the C language APIs listed in "Perform operations using APIs" on page 47 can operate on the "root"
(/) file system in a threadsafe manner.

| ### Journal object changes in the ″root″ (/) file system
| Objects in the ″root″ (/) file system can be journaled. 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. For
| more information on journaling object changes in the ″root″ (/) file system, see Chapter 7, "Journaling
| support for integrated file system objects" on page 87.

Open systems file system (QOpenSys)

The QOpenSys file system is compatible with UNIX-based open system standards, such as POSIX and
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/output.

- Supports multiple hard links and symbolic links.
- Supports case-sensitive names.
- Supports local sockets.
- Supports *OOPOOL objects.
- Support threadsafe APIs.
- Supports *FIFO objects.
- Supports the journaling of object changes.

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.

For more information on QOpenSys, see Use QOpenSys.

| For information about *TYPE 1 to *TYPE2 directory conversions and QOpenSys file system restrictions, | see *TYPE2 directories.

Use QOpenSys

QOpenSys can be accessed through the integrated file system interface using either the OS/400 file server or the integrated file system commands, user displays, and C language APIs.

Case-sensitivity in the QOpenSys file system

Unlike the "root" (/) file system, the QOpenSys file system distinguishes between uppercase or lowercase 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 `Payroll`, an object named `PayRoll`, and an object named `PAYROLL` in the same directory in QOpenSys.

Path names in the QOpenSys file system

- Path names have the following form:

      `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 megabytes 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 (see "Name continuity" on page 22). Refer to *TYPE2
| directories for more information about the directory formats.

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.

See "Link" on page 18 for a description of links.

Use integrated file system commands and displays in the QOpenSys file system

All of the commands that are listed in "Perform operations using CL commands" on page 26 and the displays that are described in "Perform operations using iSeries menus and displays" on page 25 can operate on the QOpenSys file system. However, it may not be safe to use these commands in a multi-thread capable process.

Use integrated file system APIs in the QOpenSys file system

All of the C language functions that are listed in "Perform operations using APIs" on page 47 can operate on the QOpenSys file system in a threadsafe manner.

Journal object changes in the QOpenSys file system

Objects in the QOpenSys file system can be journaled. 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. For more information on journaling object changes in the QOpenSys file system, see Chapter 7, "Journaling support for integrated file system objects" on page 87.

User-defined file system (UDFS)

The UDFS file systems reside on the auxiliary storage pool (ASP) or independent auxiliary storage pool (ASP) of your choice. You 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/output.
- Support multiple hard links and symbolic links.
- Support local sockets.
- Support threadsafe APIs.
- Support *FIFO objects.
- Support the journaling of object changes.

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.

See the following topics for more information about user-defined file systems:

- UDFS concepts
- Use UDFS through the integrated file system interface

UDFS concepts

In a UDFS, as in the "root" (/) and QOpenSys file systems, you can create directories, stream files, symbolic links, local sockets, and SOM 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 sub-directories, 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.

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

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

Use UDFS through the integrated file system interface

A UDFS can be accessed through the integrated file system interface using either the OS/400 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.

Case-sensitivity in an integrated file system UDFS

You can specify whether object names in the UDFS will be 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 CRTUDFS command.

When you select case-insensitivity, the server does not distinguish between uppercase and lowercase characters during searches for names. Therefore, the server would recognize /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 server.

Path names in an integrated file system UDFS

A block special file (*BLKSF) represents a 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 megabytes 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 (see "Name continuity" on page 22). Refer to *TYPE2
| directories for more information about the directory formats.

Links in an integrated file system UDFS

Objects within a 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.

See "Link" on page 18 for a description of links.

Use integrated file system commands in a UDFS

All of the commands that are listed in "Perform operations using CL commands" on page 26 and the displays that are described in "Perform operations using iSeries menus and displays" on page 25 can operate on a user-defined file system. There are some CL commands that are specific to the user-defined file system and other mounted file systems in general. The following table describes them.

Table 6. 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.

Use integrated file system APIs in a UDFS

All of the C language functions that are listed in "Perform operations using APIs" on page 47 can operate on a user-defined file system.

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

Graphical user interface for a UDFS

iSeries Navigator, a graphical user interface on your PC, provides for easy and convenient access to 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:
• "Create a new user-defined file system" on page 41.
• "Mount a user-defined file system" on page 41.
• "Unmount a user-defined file system" on page 42.

Create an integrated file system UDFS

The Create User-Defined File System command (CRTUDFS) 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 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 case-sensitivity.

Delete an integrated file system UDFS

The Delete User-Defined File System command (DLTUDFS) deletes an existing, unmounted 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 proper authority to delete all of the objects within a UDFS, then none of the objects will be deleted.

Display an integrated file system UDFS

The Display User-Defined File System (DSPUDFS) command presents the attributes of an existing 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.

Mount an integrated file system UDFS

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 UDFS, you need to specify *UDFS for the TYPE parameter on the ADDMFS command.

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

Unmount an integrated file system UDFS

The unmount command makes the contents of a 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 are 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 can not be mounted over.

Save and restore an integrated file system UDFS

You have the ability to save and restore all UDFS objects, as well as their associated authorities. The Save command (SAV) allows you to save objects in a UDFS while the Restore command (RST) allows you to restore UDFS objects. Both commands will 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.

Journal object changes in a UDFS file system

Objects in user-defined file systems can be journaled. 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. For more information on journaling object changes in a UDFS file system, see Chapter 7, "Journaling support for integrated file system objects" on page 87.

Library file system (QSYS.LIB)

The QSYS.LIB file system supports the iSeries server library structure. This file system provides you with access to database files and all of the other iSeries server object types that the library support manages in the system and basic user ASPs.

In addition, it:

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

Before Version 3 of OS/400, the QSYS.LIB file system would likely have been known as *the* iSeries server 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.

For more information on QSYS.LIB, see Use QSYS.LIB through the integrated file system interface.

Use 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 OS/400 file server or the integrated file system commands, user displays, and C language APIs. In using the integrated file system interfaces, you should be aware of the following considerations and limitations.

QPWFSERVER authorization list in the QSYS.LIB file system

The QPWFSERVER 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

- 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() topic in the iSeries Information Center.)
- 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/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 server. They may, 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 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 CCSID, see the Globalization topic in the iSeries Information Center.

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.

See "Link" on page 18 for a description of links.

Use integrated file system commands and displays in the QSYS.LIB file system

The commands listed in "Perform operations using CL commands" on page 26 can operate on the QSYS.LIB file system, except for the following:

- 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 can not be used on database physical files.

The same restrictions apply to the user displays described in "Perform operations using iSeries menus and displays" on page 25.

Use integrated file system APIs in the QSYS.LIB file system

The C language functions listed in "Perform operations using APIs" on page 47 can operate on the QSYS.LIB file system, except for the following:

- 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.

Independent ASP QSYS.LIB

The Independent ASP QSYS.LIB file system supports the iSeries server 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 iSeries server object types that the library support manages in the independent ASPs.

In addition, it:

- Supports all user interfaces and programming interfaces that operate on iSeries server 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 iSeries server objects
- Supports stream I/O operations on physical file members, user spaces, and save files

For more information on the Independent ASP QSYS.LIB file system, see Use Independent ASP QSYS.LIB through the integrated file system interface.

Use 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 OS/400 file server or the integrated file system commands, user displays, and C language APIs. In using the integrated file system interfaces, you should be aware of the following considerations and limitations.

QPWFSERVER authorization list in the Independent ASP QSYS.LIB file system

The QPWFSERVER 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

- 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() topic in the iSeries Information Center.)
- 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/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 server. 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 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 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 Globalization topic in the iSeries Information Center.

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.

See "Link" on page 18 for a description of links.

Use integrated file system commands and displays in the Independent ASP QSYS.LIB file system

The commands listed in "Perform operations using CL commands" on page 26 can operate on the Independent ASP QSYS.LIB file system, except for the following:

- 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 can not be used on database physical files.
- You can not 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, 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 same restrictions apply to the user displays described in "Perform operations using iSeries menus and displays" on page 25.

Use integrated file system APIs in the Independent ASP QSYS.LIB file system

The C language functions listed in "Perform operations using APIs" on page 47 can operate on the Independent ASP QSYS.LIB file system, except for the following:

- 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.

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 iSeries server folders and document library objects (DLOs).
- Supports data stored in stream files.

For more information on QDLS, see Use QDLS through the integrated file system interface.

Use QDLS through the integrated file system interface

The QDLS file system can be accessed through the integrated file system interface using either the OS/400 file server or the integrated file system commands, user displays, and C language 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 known as HFS. Whereas the integrated file system is based on the integrated language environment (ILE) program model, HFS is based on the original iSeries server 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.

For more information about HFS, see the Hierarchical File System APIs topic in the iSeries Information Center.

User enrollment in the QDLS file system

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

Case-sensitivity in the QDLS file system

QDLS 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.

For more details, see the Folder and Document Name topic in the iSeries Information Center.

Path names in the QDLS file system

- Each component of the path name can consist of just a name, such as:

        /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 QDLS 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 Globalization topic in the iSeries Information Center.

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.

See "Link" on page 18 for a description of links.

Use integrated file system commands and displays in the QDLS file system

The commands listed in "Perform operations using CL commands" on page 26 can operate on the QDLS file system, except for the following:

- 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.
- The APYJRNCHG, ENDJRN, SNDJRNE, and STRJRN commands are not supported.

The same restrictions apply to the user displays described in "Perform operations using iSeries menus and displays" on page 25.

Use integrated file system APIs in the QDLS file system

The C language functions listed in "Perform operations using APIs" on page 47 can operate on the QDLS file system, except for the following:

- The symlink() function can be used only to link <u>to</u> an object in QDLS from another file system that supports symbolic links.
- The following functions are not supported:

    givedescriptor()

    ioctl()

    link()

    QjoEndJournal()

    QjoRetrieveJournalInformation()

    QJORJIDI()

    QJOSJRNE()

    QjoStartJournal()

    Qp0lGetPathFromFileID()

    readlink()

    takedescriptor()

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/output.
- Supports data stored in stream files.

For more information on QOPT, see Use QOPT through the integrated file system interface.

Use QOPT through the integrated file system interface

The QOPT file system can be accessed through the integrated file system interface using either the OS/400 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.

For more details, see the publication Optical Support .

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 known as HFS. Whereas the integrated file system is based on the integrated language environment (ILE) program model, HFS is based on the original iSeries server 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 on HFS APIs, see the hierarchical file system APIs topic in the iSeries Information

Center, or in the publication Optical Support .

Case-sensitivity in the QOPT file system

Depending upon 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 sub-directory 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 simply 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

the publication Optical Support .

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.

See "Link" on page 18 for a description of links.

Use integrated file system commands and displays in the QOPT file system

Most commands listed in "Perform operations using CL commands" on page 26 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 "Perform operations using iSeries menus and displays" on page 25.

The following integrated file system commands are not supported by the QOPT file system:
- ADDLNK
- APYJRNCHG
- CHKIN
- CHKOUT
- ENDJRN
- SNDJRNE
- STRJRN
- WRKOBJOWN
- WRKOBJPGP

Use integrated file system APIs in the QOPT file system

All of the C language APIs listed in "Perform operations using APIs" on page 47 can operate on the "root" (/) file system in a threadsafe manner, except for the following:
- QjoEndJournal()
- QjoRetrieveJournalInformation()
- QJORJIDI()
- QJOSJRNE()
- QjoStartJournal()

NetWare file system (QNetWare)

The QNetWare file system provides access to data on a local or remote Integrated xSeries Server for iSeries running Novell NetWare 4.10 or 4.11, or to stand alone PC Servers running Novell NetWare 3.12, 4.10, 4.11 or 5.0.

In addition, it:
- 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

Note: The QNetWare file system is available only when NetWare Enhanced Integration for iSeries 400, BOSS option 25, is installed on the system. After the next IPL following the installation, the `/QNetWare` directory and its sub-directories appear as part of the integrated file system directory structure.

See the following topics for more information about the QNetWare file system:
- Mount NetWare file systems
- QNetWare directory structure
- Use QNetWare through the integrated file system interface

Mount 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 File System (ADDMFS) command, such as mounting a read-write file system as read-only.

NetWare file systems can be mounted using an NDS path or by specifying a NetWare path in the form of SERVER/VOLUME:directory/directory. For example, to mount the directory doorway located in volume Nest on server Dreyfuss, you would 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 cannot themselves 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.O/ORG_UNIT.OU/SVR1_VOL.CN

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

Use QNetWare through the integrated file system interface

The QNetWare file system can be accessed through the integrated file system interface using either the OS/400 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 iSeries server users based on the name of a user. If the NetWare name exceeds ten characters or a corresponding iSeries server 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 WRKAUT and CHGAUT commands. When authorities are transferred to and from the server they are mapped to iSeries server authorities.

Audit 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 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.

NDS objects in the QNetWare file system

The QNetWare file system supports the display of 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.

Use integrated file system commands and displays in the QNetWare file system

The commands listed in "Perform operations using CL commands" on page 26 can operate on the QNetWare file system, except for the following:

ADDLINK
| APYJRNCHG
CHGAUD
CHGPGP
CHKIN
CHKOUT
| ENDJRN
| SNDJRN
| 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

Use integrated file system APIs in the QNetWare file system

The C language functions listed in "Perform operations using APIs" on page 47 can operate on the QNetWare file system, except for the following APIs:

givedescriptor()
link()
| QjoEndJournal()
| 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()

Windows NT Server File System (QNTC)

The QNTC file system provides access to data and objects that are stored on a local or remote Integrated xSeries Server for iSeries running Windows NT 4.0 Server or higher or standalone server. It allows iSeries server applications to use the same data as Windows NT clients. It stores data in stream files.

The QNTC file system is part of the base OS/400 operating system. To use the QNTC file system, you must have TCP/IP Connectivity Utilities for iSeries 400 (part number: 5769-TC1) installed. It is not necessary to have the iSeries 400 Integration with Windows NT Server, option 29 of the Operating System, installed to access /QNTC.

For more information on QNTC, see Use QNTC through the integrated file system interface.

Use QNTC through the integrated file system interface

By using either the OS/400 file server or the integrated file system commands, user displays, and APIs, you can access the QNTC file system through the integrated file system interface. You should 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 name must begin with a slash and can be up to 255 characters long.
- Path names are case sensitive.
- The path consists of the file system name, the Windows NT server name, the sharename, the directory and sub-directory names, and the object name. Path names have the following form:

      /QNTC/Servername/Sharename/Directory/ . . . /Object
  (QNTC is a required part of the path name.)
- The server name can be up to 15 characters long. It must be part of the path.
- The sharename can be up to 12 characters long.
- Each component of the path name after the sharename 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.
- Each functional Windows NT server in the local subnet will automatically appear as a directory under /QNTC. Use the Make Directory (MKDIR) command (see Table 2 on page 26) or mkdir() API (see "Perform operations using APIs" on page 47) to add Windows NT servers outside the local subnet.

Links in the QNTC file system

The QNTC file system supports only one link to an object. You can not 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.

See "Link" on page 18 for a description of links.

Use integrated file system commands and displays in the QNTC file system

The commands listed in "Perform operations using CL commands" on page 26 can operate on the QNTC file system, except for the following:

    ADDLNK
    APYJRNCHG
    CHGOWN
    CHGAUT
    CHGPGP
    CHKIN
    CHKOUT
    DSPAUT
    ENDJRN
    RST
    SAV
    SNDJRNE
    STRJRN
    WRKAUT
    WRKOBJOWN
    WRKOBJPGP

The same restrictions apply to the user displays that are described in "Perform operations using iSeries menus and displays" on page 25.

Use the MKDIR command in the QNTC file system

Use the Make Directory (MKDIR) command to add a server directory to the /QNTC directory. All functional Windows NT servers in the local subnet are automatically created. Those Windows NT servers outside the local subnet must be added using the MKDIR command or mkdir() API. For example:

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

would add 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'
```

would add the server into the QNTC file system directory structure.

Note: 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.

Use integrated file system APIs in the QNTC file system

The C language functions listed in "Perform operations using APIs" on page 47 can operate on the QNTC file system, except for the following:

- 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()

    givedescriptor()

    link()

    QjoEndJournal()

    QjoRetrieveJournalInformation()

    QJORJIDI()

    QJOSJRNE()

    QjoStartJournal()

    Qp0lGetPathFromFileID()

    readlink()

    symlink()

    takedescriptor()

OS/400 File Server File System (QFileSvr.400)

The OS/400 file server file system provides transparent access to other file systems that reside on remote iSeries servers. 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 OS/400 file server on the target system to perform the actual file operation.

For more information on QFileSvr.400, see Use QFileSvr.400 through the integrated file system interface.

Use 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 OS/400 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 OS/400 file server 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 OS/400 file server.

Path names in the OS/400 file server file system

- Path names have the following form:

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

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

– The name of the target server that will be used to establish a communications connection. The target server name can be either of the following:

  - 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 server

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 (see "Name continuity" on page 22).

Communications in the OS/400 file server file system

- 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

  the publication 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 OS/400 system job Q400FILSVR.

- If a connection is not already 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 OS/400 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 OS/400 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 OS/400 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 OS/400 file server file system

If both of the systems have Kerberos configured, and the user has authenticated to Kerberos, then Kerberos may be used to authenticate a file system that resides on a target iSeries server. If the Kerberos authentication fails, then the user ID and password may be used to verify access.

Note: If the ticket-granting ticket or the server ticket expires after the target server has verified your access, the expiration will not be effective until the connection to the target server has ended. For more information on Kerberos, see the Network authentication service topic in the iSeries Information Center.

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

  Note: If your password on the local or target server is changed after the target server has verified your access, then the change is not reflected until the connection to the target server has ended. However, there is no delay if your user profile on the local server 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 server.

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

Links in the OS/400 file server 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.

See "Link" on page 18 for a description of links.

Use integrated file system commands and displays in the OS/400 file server file system

The commands listed in "Perform operations using CL commands" on page 26 can operate on the QFileSvr.400 file system, except for the following:

ADDLNK

APYJRNCHG

CHGAUT

CHGOWN

DSPAUT

| ENDJRN

RST

SAV

| SNDJRNE

| STRJRN

WRKOBJOWN

WRKOBJPGP

The same restrictions apply to the user displays described in "Perform operations using iSeries menus and displays" on page 25.

Use integrated file system APIs in the OS/400 file server file system

The C language functions listed in "Perform operations using APIs" on page 47 can operate on the QFileSvr.400 file system, except for the following:

chown()

fchown()

givedescriptor()

link()

| QjoEndJournal()

| QjoRetrieveJournalInformation()

| QJORJIDI()

| QJOSJRNE

| QjoStartJournal

Qp0lGetPathFromFileID()

symlink()

takedescriptor()

Network File System (NFS)

The NFS file system 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.

For more information on NFS, see Use NFS file systems through the integrated file system interface.

Use NFS file systems through the integrated file system interface

The Network File System is accessible through the integrated file system interface and has the following considerations and limitations.

Characteristics of the network file system

The characteristics of any file system mounted through NFS are dependent upon 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 would appear to be an extension of the local directory, it would actually function and perform 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 will function and what its characteristics will be:

1. The user mounts a file system from an iSeries server on a client.

2. The user mounts a file system from a UNIX server on a client.

3. The user mounts a file system from a non-iSeries, non-UNIX server on a client.

In the first scenario, the mounted file system will behave on the client similarly to how it behaves on the iSeries server. 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 will have the characteristics and limitations of the QDLS file system. For example, in the QDLS file system, pathname components are limited to 8 characters plus a 3 character extension. However, the mounted file system will also have NFS characteristics and limitations. For example, you can not 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 will behave most similar to the iSeries server QOpenSys file system. To find out more about the QOpenSys file system, please see "Open systems file system (QOpenSys)" on page 61.

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

See "Link" on page 18 for a description of links.

Use integrated file system commands in the network file system

All of the commands that are listed in "Perform operations using CL commands" on page 26 and the displays that are described in "Perform operations using iSeries menus and displays" on page 25 can operate on the Network File System, except for the following:

- APYJRNCHG
- CHGAUD
- CHGATR
- CHGAUT
- CHGOWN
- CHGPGP
- CHKIN
- CHKOUT
- ENDJRN

| • 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 7. 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.

Use integrated file system APIs in the network file system

| All of the C language functions that are listed in "Perform operations using APIs" on page 47 can operate
| on the network file system, except for the following:
| • QjoEndJournal()
| • 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.

Chapter 7. Journaling support for integrated file system 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.

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.

The following topics introduce journaling support for integrated file system objects:
- "Journal management"
- "Objects you should journal"
- "Journaled integrated file system objects" on page 88
- "Journaled operations" on page 89
- "Special considerations for journal entries" on page 89

For detailed information on journaling integrated file system objects, refer to the Journal management topic in the iSeries Information Center.

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
- Assistance in testing application programs

You can use a journal to define what objects you want to protect with journal management. See "Objects you should journal" for more considerations on journaling objects. 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.

Objects you should journal

Consider the following questions when deciding if you should journal an integrated file system object:
- 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 would it be 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 would be 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 would the delay in reconstructing changes have on the business?
- How does the object relate to other objects on the server? Although the data in a particular object may not change often, that object's data may be critical to other, more dynamic objects on the server. 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.

Journaled integrated file system objects

Some integrated file system object types can be journaled using OS/400 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 information.

Note: Memory mapped stream files and stream files that are used by the Integrated xSeries Server for iSeries (IXS) for virtual drive storage space can not be journaled.

Following is a list that 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, and Apply Journaled Changes, may be performed on entire subtrees of integrated file system objects. You can optionally use the include and exclude lists that may use wild card patterns for object names. For example, you could 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.

  Journaled 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 journaled 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: If you end journaling for an object, and then rename that 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.

- 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 journaled integrated file system objects.
- The maximum write limit supported by the write() and writev() APIs is 2 gigabytes-1. The maximum journal entry size if RCVSIZOPT (*MAXOPT2) is specified is 4,000,000,000. Otherwise, the maximum journal entry size is 15, 761,440 bytes. If you journal your stream file and have any writes which exceed 15, 761,440 bytes, you will want to use the *MAXOPT2 support to prevent any errors from occurring.

For more detailed information on journaling integrated file system objects, refer to Journal management in the iSeries Information Center.

For more information about the layout of various journal entries, there is a C language include file, qp0ljrnl.h, shipped in member QSYSINC/H (QP0LJRNL), that contains details of the integrated file system journal entry specific data content and formats.

For a complete list of all journal entries deposited for integrated file system objects, refer to the Journal code finder in the iSeries Information Center.

Journaled operations

The following operations are only journaled when the type of the object or link that the operation is using is a type that can also be journaled:

- 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 journaled operations are specific to a stream file:

- Data write
- File truncate/extend
- File data forced
- Save with storage freed

The following journaled operations apply to all journaled object types:

- Attribute changes (including security changes such as authorities and ownership)
- Open
- Close
- Start journaling
- End journaling
- Start the Apply Journaled Changes (APYJRNCHG) command
- End the Apply Journaled Changes (APYJRNCHG) command
- Save
- Restore

For more detailed information on journaling integrated file system objects, refer to the Journal management topic in the iSeries Information Center. For a complete list of all journal entries deposited for integrated file system objects, refer to the Journal code finder in the **Systems management** topic.

Special considerations for journal entries

Many journaled 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)
- 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 and/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.

Appendix A. Transport Independent Remote Procedure Call

Developed by Sun Microsystems, Remote Procedure Call (RPC), easily separates and distributes client applications from a server mechanism. It includes a standard for data representation, called eXternal Data Representation, or XDR, to allow 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 iSeries server 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 towards distributing the user interface or data retrieval.

Network selections

The following 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
getnetconfigent()	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.

Name-to-address translation

The following 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

API	Description
uaddr2taddr()	Translates a transport-independent (universal) address to a transport-specific (local) address (netbuf structure)

eXternal Data Representation (XDR)

The following APIs allow Remote Procedure Call (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.
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()	Afilter 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.

API	Description
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
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.

Authentication

The following APIs provide authentication to the Transport-Independent Remote Procedure Call (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

Transport Independent RPC (TI-RPC)

The following APIs provide a distributed application development environment by isolating the application from any specific transport feature. This adds ease-of-use to the transports.

TI-RPC simplified APIs

The following 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

TI-RPC top-level APIs

The following 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
clnt_destroy()	Destroy the client's RPC handle
svc_create()	Create a server handle
svc_destroy()	Destroy an RPC service transport handle

TI-RPC intermediate level APIs

The following 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

TI-RPC expert level APIs

The following 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()

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

API	Description
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
svcerr_systemerr()	Sends information to client for system error

Appendix B. Example program using 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.

This disclaimer information pertains to code examples.

```
#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  ".."
```

```c
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;
}
```

Appendix C. Integrated file system RPG code example

Code Snippets contains an integrated file system RPG code example. To view this example, perform the following steps:

1. Select **ILE RPG Source** from the drop-down list in the Search category.
2. Click **Search**.
3. Scroll down the list until you see **Using IFS from RPG**.
4. Click **Code for using IFS from RPG**.

This disclaimer information pertains to code examples.

Bibliography

This bibliography lists iSeries server information that contains background information or more details for information discussed in this book.

- The Control language topic in the **Programming** category of the iSeries Information Center provides a description of the iSeries server control language (CL) and its commands. Each command description includes a syntax diagram, parameters, default values, keywords, and an example.
- The Globalization topic in the iSeries Information Center explains national language support (NLS) concepts, such as character set and code page, and provides information needed to evaluate, plan, and use the iSeries server NLS and multilingual capabilities.
- The APIs topic in the **Programming** category of the iSeries Information Center provides a description of each OS/400 API, including the integrated file system APIs.
- The Journal management topic in the **Systems management** category of the iSeries Information Center provides information on how to set up, manage, and troubleshoot system-managed access-path protection (SMAPP), local journals, and remote journals on an iSeries server.
- The Commitment control topic in the **Database** category of the iSeries Information Center category explains how to define and process a group of changes to resources, such as database files or integrated file system files, as a logical unit of work.

- OS/400 Network File System Support 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.

- Optical Support This book serves as a user's guide and reference for IBM Optical Support on OS/400. The information in this book can help the user understand optical library dataserver concepts, plan for an optical library, administer and operate an optical library dataserver, and solve optical dataserver problems.

- WebSphere Development Studio: ILE C/C++ Programmers Guide This book provides information needed to design, edit, compile, run, and debug ILE C/400 programs on the iSeries server.
- WebSphere Development Studio: C/C++ Language Reference This book provides information about the structure of ILE C/400 programs and contains details on the library functions and include (header) files.

- Security — Reference This book provides detailed technical information about OS/400 security.

- APPC Programming This book describes the advanced program-to-program communications (APPC) support for the iSeries server. It guides in developing application programs that use APPC and in defining the communications environment for APPC.

- Backup and Recovery This book provides general information about recovery and availability options for the IBM iSeries server.

Bibliography

Index

A

absolute path name   17
access mode   54
API
   example program   97
   ILE C/400   52
   path name rules   53
   using in C programs   43, 47
authority
   commands   26
   handling in programs   54
   in example program   97
   limitations for QFileSvr.400 file system   82
   limitations for QNTC file system   78

B

bibliography   105
binary open file mode   56

C

C language program
   example   97
   ILE C/400 functions   52
character conversion   2, 22, 55
Client Access   32
code page   2, 22, 55
commands
   list   26
   path name rules   29
   using   26
conversion
   data   56
   object names   22, 55
current directory   9

D

data conversion   56
database file
   comparison with stream file   3
   copying to/from stream file   43
   creating from stream file   43
directory
   benefit   1
   commands   26
   current   9
   home   9
   in example program   97
   integrated file system   34
   menus and displays   26
   what is it?   6
displays
   path name rules   29
   using   25

document library services (QDLS) file system
   characteristics and limitations   72
   description   5

E

encoding schemes   22, 55
example
   path names   17, 29, 53
   program using integrated file system APIs   97
   using symbolic link   19
extended attributes
   continuity across national languages   22, 55
   naming guidelines   22
   what are they?   21

F

file   97
   menus and displays   26
   open modes   56
   transferring   31
file descriptor   54
file server   6
file server I/O processor   6
file system
   benefit   2
   comparison   57
   document library services (QDLS)
      characteristics and limitations   72
      description   5
   independent ASP QSYS.LIB
      description   4
   Independent ASP QSYS.LIB
      characteristics and limitations   69
   interface   5
   library (QSYS.LIB)
      characteristics and limitations   67
      description   4
   migrating objects   33
   NetWare file system (QNetWare)
      characteristics and limitations   75
   Network file system
      description   5
   Network File System
      characteristics and limitations   83
   open systems (QOpenSys)
      characteristics and limitations   61
      description   4
   optical (QOPT)
      characteristics and limitations   73
   Optical file system (QOPT)
      description   5
   OS/400 file server (QFileSvr.400)
      characteristics and limitations   80
   QFileSvr.400 file system (QFileSvr.400)
      description   5

IBM ®

Printed in U.S.A.