OS/390

UNIX System Services User's Guide

SC28-1891-09

OS/390

UNIX System Services User's Guide

SC28-1891-09

Note

Before using this information and the product it supports, be sure to read the general information under “Appendix H. Notices” on page 357.

Tenth Edition (September 2000)

This is a major revision of SC28-1891-08.

This edition applies to Version 2 Release 10 of OS/390 (5647-A01) and to all subsequent releases and modifications until otherwise indicated in new editions.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address below.

IBM welcomes your comments. A form for readers' comments may be provided at the back of this publication, or you may address your comments to the following address:

International Business Machines Corporation
Department 55JA, Mail Station P384
2455 South Road
Poughkeepsie, NY 12601-5400
United States of America

FAX (United States & Canada): 1+845+432-9405
FAX (Other Countries):
Your International Access Code +1+845+432-9405

IBMLink (United States customers only): IBMUSM10(MVHRCFS)
Internet e-mail: mhvrcfs@us.ibm.com
World Wide Web: http://www.ibm.com/s390/os390/webqs.html

If you would like a reply, be sure to include your name, address, telephone number, or FAX number.

Make sure to include the following in your comment or note:

- Title and order number of this book
- Page number or topic related to your comment

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Information in this document has been adapted from the *InterOpen™/POSIX Shell and Utilities User Manual,* supplied by Mortice Kern Systems (MKS) Inc. for use by licensees of their InterOpen/POSIX Shell and Utilities source code product.

The information contained in the glossary section and tagged by the word [POSIX] is copyrighted information of the Institute of Electrical and Electronics Engineers, Inc., extracted from IEEE Std 1003.1-1990, IEEE P1003.0, and IEEE P1003.2. This information was written within the context of these documents in their entirety. The IEEE takes no responsibility or liability for and will assume no liability for any damages resulting from the reader's misinterpretation of said information resulting from the placement and context in this publication. Information is reproduced with the permission of the IEEE.

Contents

Figures

Tables

About This Book

This book offers an introduction to the two shells available on OS/390 UNIX System Services — the OS/390 shell and the tcsh shell.

This book provides the information you need to use the OS/390 Shells and Utilities on an IBM MVS system. The Shells and Utilities and TSO/E (Time Sharing Option Extensions) provide commands for using OS/390 UNIX System Services (OS/390 UNIX).

This book helps you utilize the functions specified in the POSIX.2 standard (IEEE Std 1003.2-1992 and ISO/IEC 9945-1992 International Standard; Portable Operating System Interface [POSIX] Part 2: Shell and Utilities). For convenience, it also describes other OS/390 UNIX support services.

Who Should Use OS/390 UNIX System Services User's Guide?

This book is for application programmers, system programmers, and end users working on an MVS system and using OS/390 UNIX services or the OS/390 shells.

The OS/390 UNIX System Services home page,

`http://www.ibm.com/s390/unix/`

has a section "About OS/390 UNIX System Services" that describes the services and the concepts of the OS/390 UNIX environment.

What Is in OS/390 UNIX System Services User's Guide?

This book describes how to use the shells, the file system, and communication services. Using the book, you will be able to:

- Enter shell commands that request services from the system.
- Write shell scripts using the shell programming language; a shell script can be as powerful as a C-language program.
- Run shell scripts and C-language programs interactively (in the foreground), in the background, or in batch.
- Switch easily between the shells and TSO/E.
- Move MVS data sets into the file system, or move files from the file system into MVS data sets.
- Enter shell commands or TSO/E commands from the shell command line.
- Create or edit a file in the file system.
- Manage your file system.

For a discussion of the OS/390 UNIX shell commands, utilities, and TSO/E commands, and file formats, see *OS/390 UNIX System Services Command Reference*.

Tasks That Can Be Performed in More Than One Environment

There are some tasks that can be performed in more than one environment—in the shells, in TSO/E, or perhaps in ISPF. If the same task can be performed in more than one environment, that is noted.

Summary of Changes

Summary of Changes
for SC28-1891-09
OS/390 Version 2 Release 10

This book contains information previously presented in *OS/390 UNIX System Services User's Guide*, SC28-1891-08, which supports OS/390 Release 9.

The following summarizes the changes to that information.

New Information

- Information about a new OS/390 shell session option that allows the last command of a pipeline to be run in the current environment. See "Running a Command in the Current Environment" on page 51.
- A note on the limitations of the ISPF shell to authorize users to OMVS has been added. See 174.
- Instructions for using the built-in **make** command. See "Improving Performance of the make Utility" on page 178
- Directions for an OS/390 shell script, **skulker**. See "Deleting Files over a Certain Age" on page 215 for more information.

Changed Information

- Information and examples for running a shell command in batch have changed. See "Running a Shell Command in Batch" on page 165 for more information.
- Instructions for transferring HFS files to workstation files have been moved to "Chapter 21. Transferring Files between Systems" on page 297.

Deleted Information

- References to the Outboard Communications Server (OCS) have been removed.

This book includes terminology, maintenance, and editorial changes. Technical changes or additions to the text and illustrations are indicated by a vertical line to the left of the change.

Summary of Changes
for SC28-1891-08
OS/390 Version 2 Release 9

This book contains information previously presented in *OS/390 UNIX System Services User's Guide*, SC28-1891-07, which supports OS/390 Release 8.

The following summarizes the changes to that information.

New Information

- Chapters 5, 7, and 9 — new chapters pertaining to the tcsh shell.
- A new appendix "Appendix B. tcsh Shell Command Summary" on page 313 which lists the built-in tcsh shell commands with their functions.
- Information on Shared Library Support. See "Listing Directory Contents" on page 209 for more information.
- Support for long hard link names. See "Backing Up and Restoring Files from the Shell" on page 229 for more information.

- Information on listing process ID's. See "Listing process IDs of processes with open files" on page 233 for more information.
- Information about BPXBATSL, an alias for BPXBATCH. For more information, see "The BPXBATCH Utility" on page 161.

Changed Information

- Some information that previously pertained only to the OS/390 shell now pertains to both the OS/390 shell and the tcsh shell.
- A new script for changing the locale in the OS/390 shell. See "Examples: Changing Locale" on page 47.

Summary of Changes
for SC28-1891-07
OS/390 Version 2 Release 8

This book contains information previously presented in *OS/390 UNIX System Services User's Guide*, SC28-1891-06, which supports OS/390 Release 7.

The following summarizes the changes to that information.

New Information

- If you use the **—s** option with the **su** command you will not be prompted for a password. See "Switching to Superuser or Another ID" on page 87 for more information.
- The environment variables, **_BPX_SPAWN_SCRIPT** and **_BPX_EXEC_SCRIPT**, now recognize a magic number within the first line of an HFS file. See "Using the Magic Number" on page 122 for more information.
- Support for double-square-bracket ([[ ]]) conditional expressions has been added. See "Using test to Test Conditions" on page 132 for more information.
- Use of the **autoload** command improves performance of shell initialization by delaying function definition processing until the first use. See "Autoloading Functions" on page 137 for more information.
- The **df** shell command will show you which HFS data set contains a file. See "Finding the HFS Data Set that Contains a File" on page 196 for more information.
- The **—f** option on the **pax** and **tar** utilities now supports reading from and writing to MVS data sets. See "Backing Up and Restoring Files: The Options" on page 228 for more information.
- The **cp** and **mv** shell commands now support copying and moving data between MVS data sets and the HFS. See "Copying Data Using UNIX System Shell Commands" on page 277 for more information.
- The **TMP_VI** environment variable can contain a directory pathname that is specified by an administrator as an alternative location for temporary files. See "Using the TMP_VI Environment Variable" on page 265 for more information.

Changed Information

- In Chapter 8, the section ″Improving the Performance of Shell Scripts″ has been updated.

Summary of Changes
for SC28-1891-06
OS/390 Version 2 Release 7

This book contains information previously presented in *OS/390 UNIX System Services User's Guide*, SC28-1891-05, which supports OS/390 Release 6.

The following summarizes the changes to that information.

New Information

- New information about using the manpage online help tool can be found in Chapter 2: ″Entering a TSO/E Command from the OS/390 UNIX Shell″ and in Chapter 5: ″Using the man Command″.
- New information about how symbolic links are resolved for **exec()**, **spawn()**, and dlls when the sticky bit is on added to Chapter 11: ″Resolving a Symbolic Link in a Pathname″.
- New information about HFS file sharing added to Chapter 11: ″An Introduction to the Hierarchical File System″.
- Information about #! (magic number) added to Chapter 1: ″Porting Yourself from a UNIX or AIX Environment″.

Changed Information

- In Chapter 17, section ″Copying an Executable Module from the File System″ has been updated.
- In Chapter 13, section ″Creating an External Link″ has been updated.

Summary of Changes
for SC28-1891-05
OS/390 Version 2 Release 6

This book contains information previously presented in *OS/390 UNIX System Services User's Guide*, SC28-1891-04, which supports OS/390 Release 5.

The following summarizes the changes to that information.

Name Change

As part of the name change of OpenEdition to OS/390 UNIX System Services, occurrences of OS/390 OpenEdition have been changed to OS/390 UNIX System Services or its abbreviated name, OS/390 UNIX. OpenEdition may continue to appear in messages, panel text, and other code with OS/390 UNIX System Services.

New Information

- New BPXBATCH environment variables: **_BPX_BATCH_UMASK** and **_BPX_BATCH_SPAWN**

Changed Information

- In Chapter 4: Customizing the Shell, section ″Improving the Performance of Shell Scripts″ has been updated
- In Chapter 14: Handling Security for Your Files, section ″Temporarily Changing the User ID or Group ID during Execution″ has been updated

Summary of Changes
for SC28-1891-04
OS/390 Version 2 Release 5

This book contains information previously presented in *OS/390 User's Guide* , SC28-1891-03, which supports OS/390 Release 4.

The following summarizes the changes to that information.

New Information

- passwd shell command
- wall shell command
- whoami shell command
- OS/390 Print Server
- New locale object and source file name

Changed Information

- Locale Objects, Source Files, and Charmaps: removed Turkish locale; file name changes

Summary of Changes
for SC28-1891-03
OS/390 Version 2 Release 4

This book contains information previously presented in *OS/390 User's Guide* , SC28-1891-02, which supports OS/390 Release 3.

The following summarizes the changes to that information.

New Information

- RUNOPTS
- extattr shell command
- Displaying extended attributes

Changed Information

- Understanding shell variables
- Deciding how to submit background jobs
- The BPXBATCH utility
- Transferring files between systems

Summary of Changes
for SC28-1891-02
OS/390 Release 3

This book contains information previously presented in *OS/390 User's Guide* , SC28-1891-01, which supports OS/390 Release 2.

The following summarizes the changes to that information.

New Information

- New UCS-2 locales supported by OS/390
- The LIBPATH environment variable
- NFS Client on an OS/390 UNIX system
- Temporary File System (TFS)

Changed Information

- External links

Summary of Changes
for SC28-1891-01
OS/390 Release 2

This book contains information previously presented in *OpenEdition MVS User's Guide* , SC28-1891-00, which supports OS/390 Release 1.

The following summarizes the changes to that information.

New Information

- Using the UUCP network
- Ability to telnet directly into the shell from a workstation connected to an MVS host with TCP/IP
- The **printenv** shell command
- Power outages and the hierarchical file system (HFS)

Changed Information

- _BPX_SPAWN_SCRIPT environment variable
- Changing locale in the shell
- Using a STEPLIB DD statement in JCL—the data sets must be cataloged

Part 1. The OS/390 UNIX Shells

Chapter 1. An Introduction to the OS/390 Shells

There are two shells available for use on OS/390 UNIX System Services:

- The OS/390 shell
- The tcsh shell

The OS/390 shell is modeled after the UNIX System V shell with some of the features found in the KornShell. As implemented for OS/390 UNIX services, this shell conforms to POSIX standard 1003.2, which has been adopted as ISO/IEC International Standard 9945-2: 1992.

The tcsh shell is an enhanced but compatible version of csh, the Berkeley UNIX C shell. It is a command language interpreter usable as a login shell and as a shell script command processor.

Figure 1 shows how these shells fit into MVS.

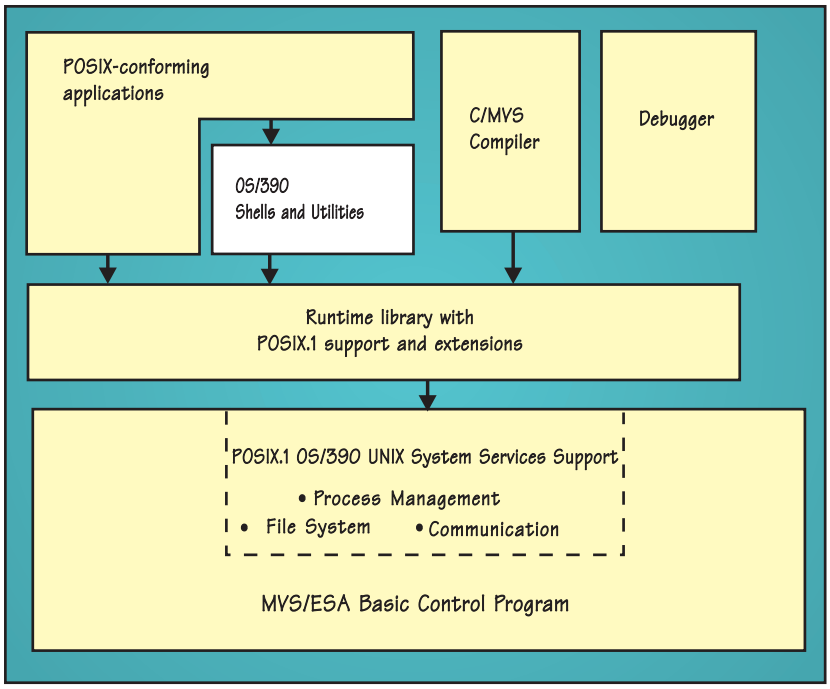


Figure 1. How the Shells Fit into MVS

About Shells

A shell is a command interpreter that you use to:

- Invoke shell commands or utilities that request services from the system.
- Write shell scripts using the shell programming language.
- Run shell scripts and C-language programs interactively (in the foreground), in the background, or in batch.

Shell Commands and Utilities

Both the OS/390 shell and the tcsh shell provide commands and utilities that give the user an efficient way to request a range of services. In this book, the term

command is used to include both a *command* (a directive to a shell to perform a specific task) and a *utility* (the name of a program callable by name from a shell).

Shell commands often have *options* (also known as *flags*) that you can specify, and they usually take an *argument*—such as the name of a file or directory. The format for specifying the command begins with the command name, then the option or options, and finally the argument, if any. For example:

```
ls -a myfiles
```

ls is the command name, **-a** is the option, and *myfiles* is the argument.

This book describes various commands you can use to perform certain tasks; most of these are shell commands, and some are TSO/E commands. Typically, this discussion highlights only certain functions of the command. For complete information about each command and all its options, always refer to *OS/390 UNIX System Services Command Reference* .

Appendix A lists OS/390 UNIX commands and utilities by the task a user might want to perform. Similar tasks are organized together.

The Locale in the Shell

A *locale* specifies cultural and language characteristics of the OS/390 UNIX System Services environment for an application program. Locale affects collation, date and time conventions, numeric and monetary formats, program messages, yes and no prompts, and the hexadecimal encoding for the 13 “variant” characters whose encoding varies on different EBCDIC code pages.

The shell and utilities support a variety of locales. See “Changing the Locale in the Shell” on page 45 for information about changing the locale in the shell.

Daemon Support

OS/390 UNIX System Services provides daemons, such as **cron**, a batch scheduler, and **inetd**, which handles **rlogin** requests.

- For information about each daemon that OS/390 UNIX System Services provides, see *OS/390 UNIX System Services Command Reference* .

Running an X-Window Application

If you are accessing the a shell from a workstation or X-terminal running an X-Window server, you can run an X-Window application from the shell. An X-Window application needs the TCP/IP address and display identifier for your workstation.

For more information on X-Window interfaces, see *TCP/IP for MVS: Programmer’s Reference*.

The Shell User

There are two categories of shell user: *superuser* and *user*. The superuser can do anything a user can, but has special authority to perform certain additional tasks (such as mounting and unmounting a file system), and can access all OS/390 UNIX services and the files in the hierarchical file system.

Security

This book assumes that your system includes the RACF security product. Instead of RACF, your system could have an equivalent security product.

The system programmer defines a shell user by assigning the user an *OMVS user ID (UID)* and *group ID (GID)*. These are numeric values associated with a TSO/E user ID; they are set in the RACF user profile and group profile when a user is authorized to use OS/390 UNIX services. The system uses the UID and GID to identify the files that a user owns and the processes that a user runs. The UID identifies a user of OS/390 UNIX services. The GID is a unique number assigned to a group of related users.

As a user, you can control read, write, and execute access to your files by other users in your group or outside of your group, by setting the permission bits associated with the files.

Accessing the Shells — The Choices

User's settings are initially configured with the OS/390 shell as the default login shell. To display these settings, from TSO type:

```
LISTUSER USERNAME OMVS
```

This will display the user's RACF settings as follows:

```
 UID= 0000000012
 HOME= /shut/home/billyjc
 PROGRAM= /bin/sh
 CPUTIMEMAX= NONE
 ASSIZEMAX= NONE
 FILEPROCMAX= NONE
 PROCUSERMAX= NONE
 THREADSMAX= NONE
 MMAPAREAMAX= NONE
 READY
```

The PROGRAM line refers to the User's login shell. If it is /bin/sh, the login shell is set to the OS/390 shell. If it is /bin/tcsh, the login shell is the tcsh shell. To change a user's default login shell from the OS/390 shell to the tcsh shell, issue the following command:

```
ALTUSER USERNAME OMVS(PROGRAM('/bin/tcsh'))
```

To change a user's default login shell from the tcsh shell to the OS/390 shell, type:

```
ALTUSER USERNAME OMVS(PROGRAM('/bin/sh'))
```

Terminal Emulators

OS/390 provides several terminal emulators that you can use to access the shells:

- The TSO/E OMVS command, a 3270 terminal interface
- The **rlogin** command, an asynchronous terminal interface
- The **telnet** command, an asynchronous terminal interface

Additionally, with OS/390 Communication Servers support, you have the asynchronous terminal interface if you directly login to the OS/390 shells from a terminal attached to a serial port on a RISC System/6000 running AIX V4.1.

When selecting a terminal emulator, there are several key points to consider:

- **Code Page Conversion:** By default, OS/390 UNIX System Services operates in the POSIX locale (also known as the C locale) using code page IBM-1047, but it can operate in other locales, including doublebyte locales. Unless you change the locale in the shell so that the code page used by the shell matches the code page used by the workstation for the OS/390 UNIX session, a terminal emulator must perform some code page conversion. Mechanisms are provided to specify the conversion required for your situation:
  - The OMVS command has the CONVERT parameter to specify the conversion between the code page used at your workstation and the code page used in the shell.
  - **rlogin** and **telnet** convert from ASCII ISO8859-1 to EBCDIC IBM-1047 by default. Once you are logged in to the shell, you can use the **chcp** to select other code pages to convert between for the session.
- **Number of Sessions:** Some terminal emulators allow multiple interactive sessions for the same user. This can be accomplished by multiple logins or by using an emulator that allows multiple sessions with one login.
- **File Editing:** With the OMVS emulator, you can use the ISPF editor. For the other terminal emulators, **vi** is the editor of choice.
- **Shell Mode: rlogin** and **telnet** provide both line mode (also known as canonical mode) and raw mode, while OMVS operates in line mode only. Line mode is sufficient for most shell utilities. However, the full function of certain useful utilities, such as **vi** and the command line editing feature of the shell, are available only in raw mode.

  When you first login to the shell, you are in line mode. Depending on your means of access, you may then be able to use utilities that require raw mode or run an X-Window application.

  line mode
  Your input is processed after you press <Enter>.

  raw mode
  Each character is processed as you type it.

  graphical mode
  A graphical user interface for X-Window applications

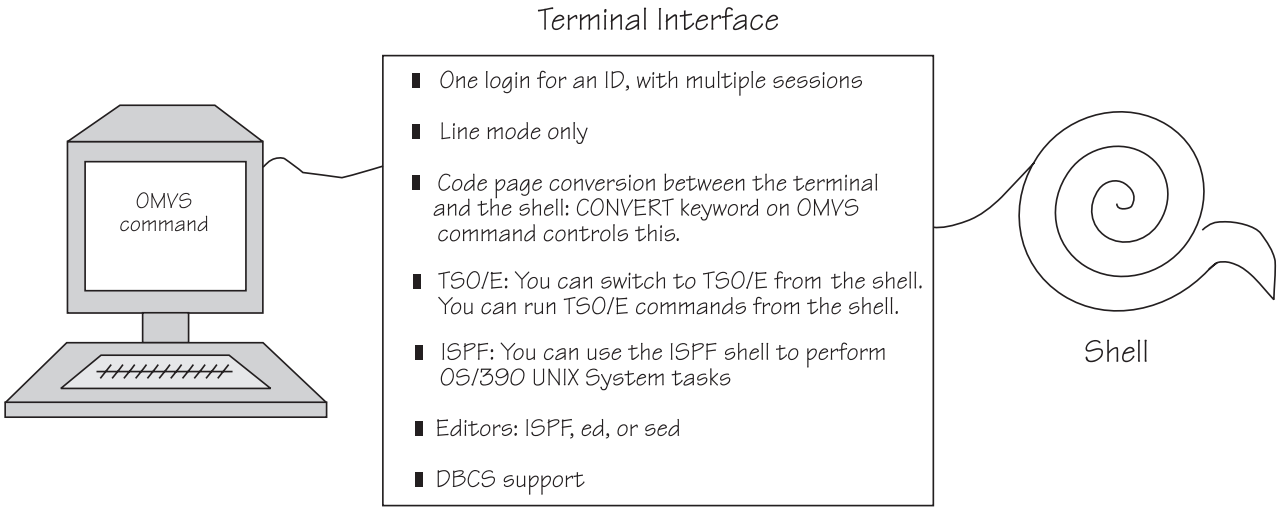


Figure 2. The OMVS Interface to the Shell

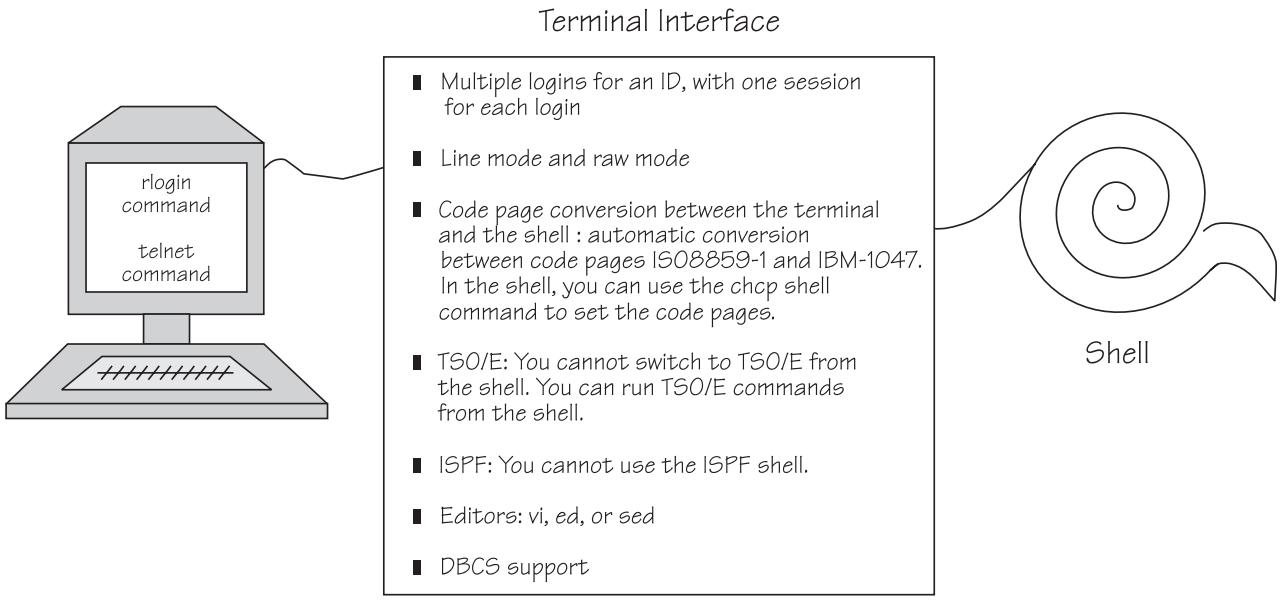


Figure 3. The Asynchronous Terminal Interface to the Shell

Interoperability between the Shells and MVS

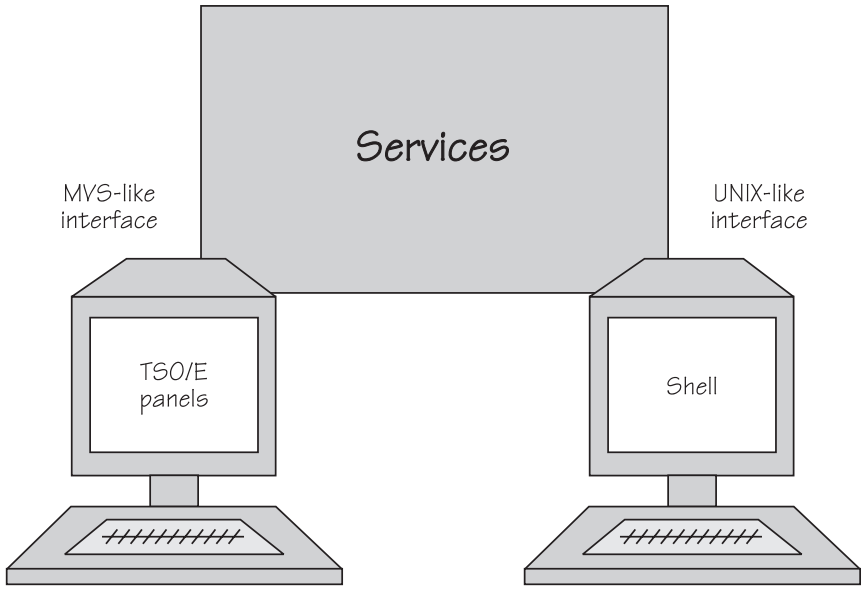


Figure 4. OS/390 UNIX System Services Provides the User Interfaces of Both MVS and UNIX

There is a high degree of interoperability between MVS and the OS/390 shells:

- You can move data between MVS data sets and the *hierarchical file system (HFS)*: You can copy or move MVS data sets into the file system; likewise, you can copy or move HFS files into MVS data sets.

- To work with the HFS, you can use TSO/E commands or shell commands. If you have access to ISPF, you can use the panel interface of the ISPF shell. For example, you can create a directory with the TSO/E MKDIR command, or the shell **mkdir** command, or the ISPF shell.
- You can issue TSO/E commands from the shell command line, from a shell script, or from a program. See “Using Commands to Work with Directories and Files” on page 201 for a list of TSO/E commands you can use to work with the file system.
- You can write MVS job control language (JCL) that includes shell commands.
- To edit HFS files, you can use the ISPF/PDF full-screen editor or one of the editors available in the shell.
- OS/390 UNIX extensions to the Restructured Extended Executor (REXX) language enable REXX programs to access kernel callable services. You can run REXX programs using these extensions from TSO/E, batch, the shell, or a C program.
- You can use the OSHELL REXX exec to run a non-interactive shell command or shell script from the TSO/E `READY` prompt and display the output to your terminal. This exec is shipped with OS/390 UNIX services.

Parallels between the MVS Environment and the Shell Environment

Figure 5 on page 9 indicates how basic programming tasks are performed in the MVS environment and in the shell environment.

An interactive user who uses the OMVS command to access the shell can switch back and forth between the shell and TSO/E, the interactive interface to MVS.

- Programmers whose primary interactive computing environment is a UNIX or AIX workstation find the shell programming environment familiar.
- Programmers whose primary interactive computing environment is TSO/E and ISPF can do much of their work in that environment.

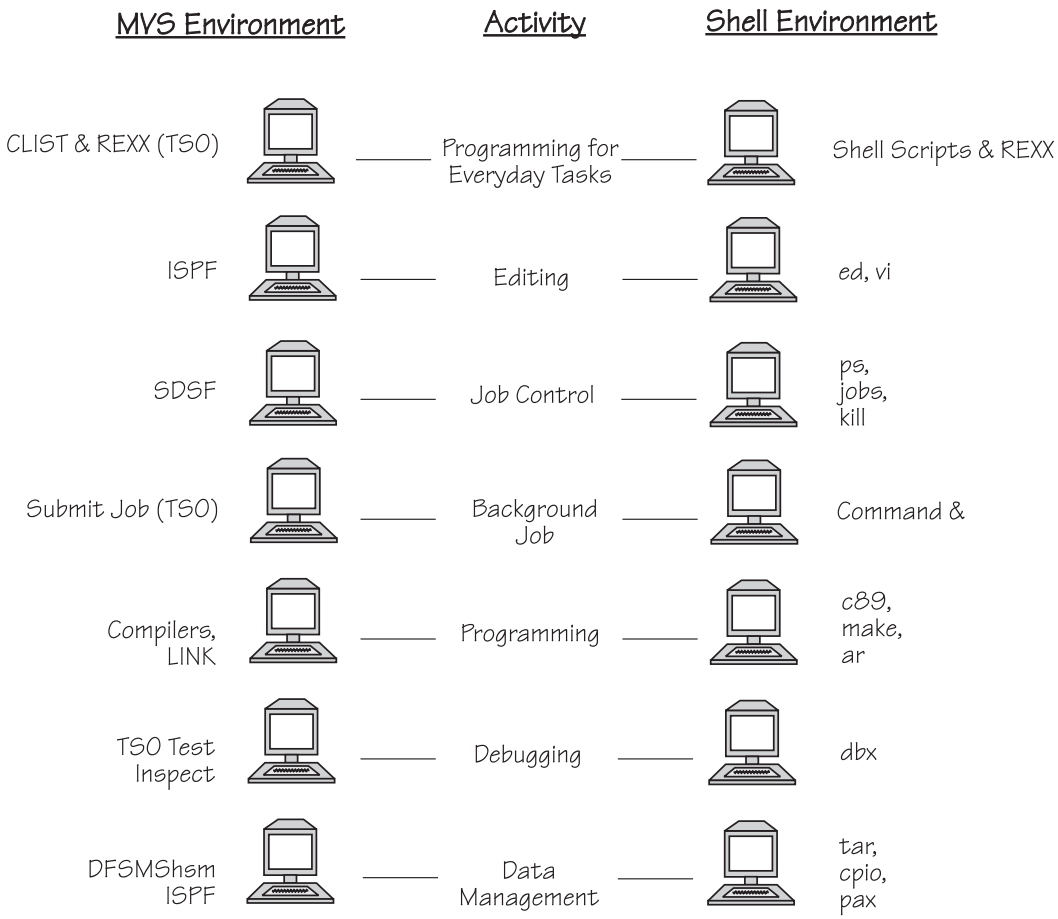


Figure 5. Working Interactively in the MVS and Shell Environments

Programming for Everyday Tasks

The shell programming environment with its shell scripts provides function similar to the TSO/E environment with its command lists (CLISTs) and the *REstructured eXtended eXecutor (REXX)* execs.

The *CLIST language* is a high-level interpreter language that lets you work efficiently with TSO/E. A *CLIST* is a program, or command procedure, that performs a given task or group of tasks. CLISTs can handle any number of tasks, from running multiple TSO/E commands to running programs written in other languages. CLISTs can run only in a TSO/E environment. For a discussion of CLISTs, see *OS/390 TSO/E CLISTs*.

The *REXX language* is a high-level interpreter language that enables you to write programs in a clear and structured way. You can use the REXX language to write programs called *REXX programs,* or *REXX execs,* that perform given tasks or groups of tasks. REXX programs can run in any MVS address space. You can run REXX programs that call OS/390 UNIX services in TSO/E, batch, in the shell environment, or from a C program. For more information about writing REXX programs, see *OS/390 TSO/E REXX User's Guide*, *OS/390 TSO/E REXX Reference*, and *OS/390 Using REXX and OS/390 UNIX System Services*.

In the shells, command processing is similar to command processing for CLISTs. You can write executable *shell scripts* (a sequence of shell commands stored in a text file) to perform many programming tasks. They can run in any dubbed MVS address space. They can be run interactively, using **cron**, or using BPXBATCH. With its commands and utilities, the shell provides a rich programming environment.

Editing

In MVS, you edit the hierarchical file system (HFS) files by using the TSO/E OEDIT command to invoke ISPF File Edit or by selecting File Edit on the ISPF menu, if installed.

In a shell, you can use the **ed** and **sed** editors for editing HFS files. You can use the **oedit** shell command to invoke ISPF File Edit. If you use **rlogin** or **telnet** to login to the shell, you can also use the **vi** editor.

Job Control

In MVS, you can use the System Display and Search Facility (SDSF) to monitor and control a job. You can also use the TSO/E CANCEL, STATUS, and OUTPUT commands.

In the shell, you use the **ps** command or the **jobs** command to check the status of a job, and the **kill** command to end a job before it completes.

Additionally, in the shell you can stop, or suspend, a foreground job, and then enter the **bg** command to run it in the background or the **fg** command to start it back up in the foreground.

Background Jobs

In MVS, you write a background job in job control language (JCL) and start it with the TSO/E SUBMIT command.

In the shell, you start a background job by typing an ampersand (**&**) at the end of the command line.

Programming

In MVS, you use the OS/390 c/c++ compiler and the linkage editor to create a traditional OS/390 c/c++ application program as a load module or to create an OS/390 c/c++ application program as an executable file or a load module.

In the shell, you can use the **c89** or **cc** or **c++** command to compile and link-edit an OS/390 UNIX program, creating an executable file. The **make** command is available for building applications, and **lex** and **yacc** are available for developing applications.

Debugging

Under TSO/E, for traditional OS/390 c/c++ application programs, TSO/E Test and Inspect facilities are available for debugging. You can use TSO/E TEST for OS/390 UNIX application programs that do not use **fork()** or **exec()**.

In the shell, **dbx** is the debugging facility for OS/390 c/c++ programs. With **dbx**, you can debug multithreaded applications at the C-source level or at the machine level. Support for multithreaded applications gives you the ability to:

- Debug or display information about the following objects related to multithreaded applications: threads, mutexes, and condition variables.
- Control program execution by holding and releasing individual threads

The **dbx** debugger provides support for recognizing, displaying, and modifying program variables and constants that include doublebyte character set (DBCS) characters.

Data Management

In MVS, the storage administrator uses Data Facility System-Managed Storage Hierarchical Storage Manager (DFSMShsm) to automatically back up and archive hierarchical file systems.

In the shell, you can use **tar**, **cpio**, and **pax** to read or write an archive file in the file system.

You can copy archive files to an MVS data set, and then to tape. You can retrieve archive files from a tape into an MVS data set and then copy them into the file system.

Chapter 2. OMVS, a 3270 Terminal Interface to the OS/390 Shell

The explanations and examples in this chapter assume that the OS/390 Shell has been set up in your profile. The information presented here is primarily directed towards users of the OS/390 shell.

The TSO/E **OMVS** command is one method of accessing the OS/390 shell. It provides a 3270 terminal interface to the shell. To use the OMVS interface to the shell, you must be working at a 3270 terminal or a computer with 3270 emulation.

You issue the **OMVS** command from TSO/E:

- In an SNA network, remote users access TSO/E through VTAM.
- In a TCP/IP network, remote users that have the Telnet 3270 client function access TSO/E by entering the TN3270 command. See the TCP/IP documentation for your system or the documentation for your computer's 3270 emulation.

For information about using an asynchronous terminal interface to the shell, see "Chapter 3. The Asynchronous Terminal Interface to the Shells" on page 35.

Differences from a UNIX or AIX Environment

If you come from a UNIX or AIX background, you will encounter some differences when you begin to use the OMVS interface to the shell. The 3270-type terminal interface may surprise you! For example:

OMVS Interface	For More Information
The 3270 interface operates in line mode (also known as canonical mode). You type data on a command line and no data is transmitted until you press the <Enter> key.	"Working in Line Mode" on page 16
The 3270 interface has function keys for various tasks such as scrolling through output, running TSO/E commands, and so on.	"Determining Function Key Settings and the Escape Character" on page 17
The OMVS interface does not have a control key. Instead of using a <Ctrl> key to type control sequences (for example, <Ctrl-D>), you use the Control function key or a multicharacter escape-key sequence.	"Typing Escape Sequences in the Shell" on page 22
With the OMVS interface, you can edit HFS files using the ISPF editor or the **ed** editor. Because this interface runs in line mode, you cannot use the **vi** editor.	"Chapter 18. Editing Files" on page 245
Delayed display of output: If a command you are running does not produce output for more than a few seconds, you will need to repeatedly press the Refresh key to display the output as it is produced.	"Why Isn't Your Output Displayed on the Screen?" on page 16

Invoking the Shell

To invoke the OS/390 shell, log on to TSO/E and then enter the TSO/E **OMVS** command. After you are working in a shell session, you can switch to TSO/E command mode or you can switch to subcommand mode.

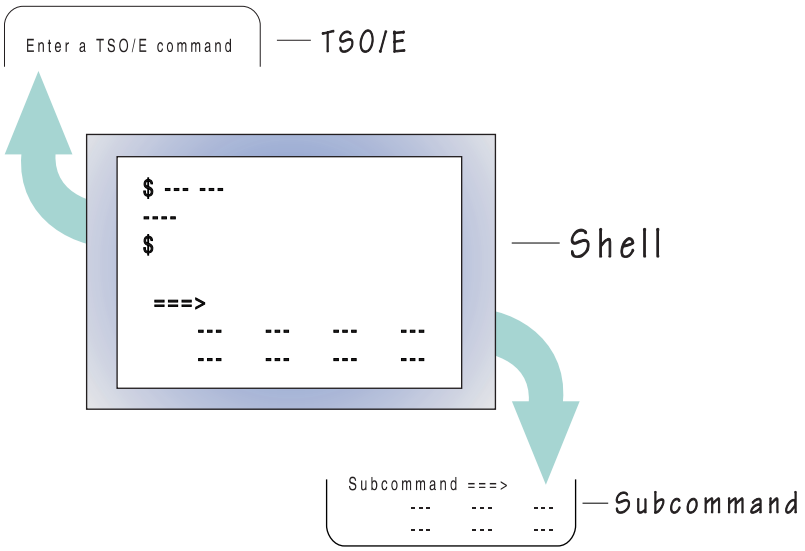


Figure 6. Switching Temporarily to TSO/E Command Mode or Subcommand Mode

To invoke the shell:

1. Log on to TSO/E with your TSO/E user ID and password.
2. At the TSO/E `READY` prompt, enter the **OMVS** command. You do not need to supply a password when invoking the shell.

You may find that the system programmer has set up your TSO/E user's logon to invoke the shell automatically; in that case, you will not need to perform step 2.

You can start multiple shell sessions simultaneously when you log into the shell, and you can start an additional shell session at any time during a shell session by using the OPEN subcommand. You can switch from session to session, using a function key or a subcommand.

Changing Options on the OMVS Command

The **OMVS** command provides an interface to the shell—for example, the layout of the screen and the processing of the function keys.

You can create a customized version of the **OMVS** command for your own use, by writing a simple REXX program or CLIST that specifies certain keywords on the command. For information on how to do this, see "Customizing the OMVS interface" on page 26 and "An Example of Customizing the OMVS Command" on page 27.

Understanding the Shell Screen

When you start the shell, you see the panel in Figure 7 on page 15.

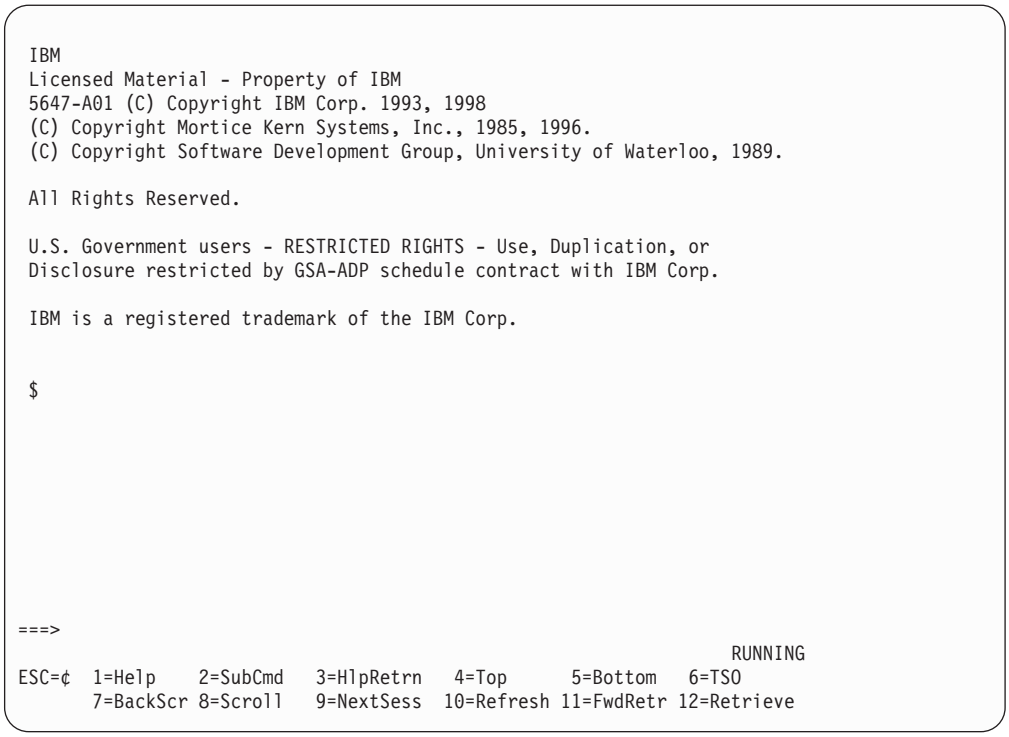


Figure 7. The OS/390 Shell's Display Screen When the Shell Is First Invoked. The bottom two lines show IBM's default function key settings.

The $ prompt is an indication from the shell that it is ready to accept input, which you type at the command line (===>). For a superuser, the default prompt is a **#**.

You can define a different prompt in your **$HOME/.profile** file, if you want to. (See Chapter 4 for more information about your **$HOME/.profile** file.)

At the bottom of the screen, you see:

- The command line (===>), used for input.
- The current function key settings and the current escape character assignments. You can turn off the function key display by typing the NOPF subcommand and turn on the display by typing the PF subcommand; alternatively, you can customize a function key to control the display of the function key settings. See "Customizing the OMVS interface" on page 26 for details on customizing function keys.

  Note: The figures in this chapter show the default function key settings.
- The status indicator in the right-hand corner, just above the function key lines. When you first enter the shell, the status indicator is RUNNING. This indicator lets you know the status of your session—for example, if an application is running or if the shell session is ready for input.
- The session number, in angle brackets, following the status indicator. The session number is displayed if there is more than one session active.

Figure 8 on page 16 shows how a screen would look after some input had been entered.

```
$ ls -l
total 7
drwxr-xr-x   2 SMITHA     0              0 Dec  3 04:25 bin
drwxr-xr-x   2 SMITHA     0              0 Nov 19 15:16 doc
-rw-rwxrwx   2 SMITHA     0            250 Nov 17 23:07 etc
-rw-r--r--   2 SMITHA     0             17 Nov 17 23:07 fora
-rw-r--r--   5 SMITHA     0           1605 Dec  3 16:38 port
-rw-r--r--   2 SMITHA     0            472 Nov 17 23:15 script
drwxr-xr-x   2 SMITHA     0              0 Nov 17 23:07 src
drwxr-xr-x  15 SMITHA     0              0 Dec  3 20:37 projecta
$









 ===>
                                                                  INPUT
ESC=¢  1=Help    2=SubCmd   3=HlpRetrn   4=Top      5=Bottom   6=TSO
       7=BackScr 8=Scroll   9=NextSess  10=Refresh 11=FwdRetr 12=Retrieve
```

Figure 8. The OS/390 Shell's Display Screen After Input Has Been Entered

At the top of the screen, `$` is the prompt and `ls -l` is the command that was entered. Beneath that is the output from the command. When a command completes, a `$` prompt is displayed, indicating you can enter another command on the command line.

If you make an error entering a command or you are running a shell script or program that ends in error, the error message is displayed in the output area. Some error messages are displayed after the last output line. Others—for example, error messages issued in subcommand mode—appear at the very top of the panel followed by a separator line. To clear an error message displayed at the top of the panel above a separator line, press <Enter> without typing any input.

Working in Line Mode

Because you are working in 3270 mode, what you type on the command line is processed in *line mode* (also known as *canonical mode*). This means your input is not processed until you press <Enter>.

- To enter input, type it at the command line (`===>`) and press <Enter>.
- To see your echoed input data or any output written by an application, look at the screen. The first line of output is displayed, and then each subsequent line of output is displayed under it.

  After the screen fills up with output lines, the older output lines scroll upward, out of view, as new output lines are displayed at the bottom of the screen. You can, however, use function keys to scroll the output backward and forward.

Why Isn't Your Output Displayed on the Screen?

After you type a command and press <Enter>, the status of your session is displayed in the lower right-hand corner of your screen as `RUNNING`. After a short time, the status indicator automatically changes to `INPUT`; this means the shell session is ready for input and will not send any more output or messages to the display screen.

At times you may find that the status indicator changes to INPUT before you have received any or all of your output. Don't worry—the shell is producing output and storing it in a buffer. Just press the Refresh function key and the shell will display more output on your screen. (If you don't have a Refresh function key, you can press a <Clear> key, <PA2>, or <PA3>.)

The reason for this behavior is that TSO/VTAM provides no way to wait for keyboard input and TTY output at the same time under TSO.

On the OS/390 UNIX System Services web site, there is some code (poll.c) that lets an OMVS user remain in RUNNING mode indefinitely. This improves usability, but it can have a significant performance impact if many people use it.

Determining Function Key Settings and the Escape Character

The shell has function keys you can use for certain tasks, instead of typing commands. To determine your function key settings and escape character assignments, look at the bottom of the screen (Figure 9).

```
ESC=¢  1=Help    2=SubCmd   3=HlpRetrn   4=Top      5=Bottom   6=TSO
       7=BackScr 8=Scroll   9=NextSess  10=Refresh 11=FwdRetr 12=Retrieve
```

Figure 9. Default Function Key Settings

The Function Key Functions

Table 1 describes *all* the functions available for function keys, and shows which of those functions are assigned by default to keys. Function keys 13 to 24 are set to the same values as function keys 1 to 12. You can change the default function key settings. For example, you may want a Control function key for typing escape sequences.

You can perform the same actions with either a function key or a subcommand; the term you see under the column "Function/Subcommand" can be entered as a subcommand also. See "Running a Subcommand" on page 25 for more information about subcommands.

In the first column, you see the default key assignment for a function; if a function is not assigned to a key by default, there is no entry in the first column. You can assign a function to a key by customizing your invocation of the **OMVS** command; see "Function Key Settings (PFn)" on page 29 for more information.

Table 1. Function Key Settings Available in the OS/390 Shell

Default Setting	Function/ Subcommand	Description
<F1> <F13>	HELP	Displays a help panel that explains the TSO/E OMVS command and the three modes you can work in: shell, subcommand, and TSO/E.

Table 1. Function Key Settings Available in the OS/390 Shell (continued)

Default Setting	Function/ Subcommand	Description
<F2> <F14>	SUBCMD	Processes an OMVS subcommand. A subcommand is a command that is passed to the OMVS command processor (instead of the shell); most subcommands are used to control, or temporarily change, the OMVS interface. You can either enter a subcommand from the shell command line or switch to subcommand mode to do it. To run a subcommand from the shell command line, type the subcommand and press this function key. To leave the shell session and enter subcommand mode, press this function key when the shell command line is empty. You can type the OMVS subcommands at the command line in subcommand mode. To resume working in the shell, press the Return function key.
<F3> <F15>	HLPRETRN or RETURN	If you are viewing help information, pressing this key removes the help information from the screen. If you are in subcommand mode, pressing this key returns you to the shell session. (Refer to "Switching to Subcommand Mode" on page 26 for a discussion of subcommand mode.)
<F4> <F16>	TOP	Scrolls displayed data back to a screenful of the oldest available output, or, in help, back to the first panel.
<F5> <F17>	BOTTOM	Scrolls displayed data to a screenful of the most recent output, or, in help, to the final panel.
<F6> <F18>	TSO	You have the choice of switching to TSO/E command mode to enter a TSO/E command or running the TSO/E command from the shell command line. To switch to TSO/E command mode, press <F6>. To resume working in the shell, press <PA1>. To run a TSO/E command from the shell command line, do either of these: • Type the command and press the TSO function key. • Use the **tso** shell command to run the TSO/E command **Note:** If you entered the OMVS command from ISPF, you cannot enter ISPF as a TSO/E command from the shell command line. You can, however, enter the ISHELL command. When the TSO/E command completes, typically *** is displayed on the screen. Press <Enter> or <Clear> to return to the shell.
<F7> <F19>	BACKSCR	Scrolls the displayed output backward, a screenful at a time. The scrolling ends when you reach the oldest available saved line in a stack of saved output lines.
<F8> <F20>	SCROLL	Scrolls the output display a full screen forward.

Table 1. Function Key Settings Available in the OS/390 Shell (continued)

Default Setting	Function/ Subcommand	Description
<F9> <F21>	NEXTSESS	Displays the next session whose session number is higher than that of the session currently displayed. However, if the highest-numbered session is currently displayed, the lowest-numbered session will be displayed.
<F10> <F22>	REFRESH	Updates the screen with new data from the shell session. Use this function key if the display of output is incomplete (for example, output from a command you issued), but the session is now displaying `INPUT` status.
<F11> <F23>	FWDRETR	Used with <F12> to retrieve commands from the stack of saved input lines. If you press <F12> one too many times and go past the line you want, you can press <F11> to display the line that was previously retrieved by <F12>.
<F12> <F24>	RETRIEVE	Retrieves the most recently entered input line from a stack of saved input lines. This key starts retrieving with the most recent in the stack of saved lines and works down to the oldest available.
	ALARM	Toggles the setting for the 3270 alarm to sound when the shell writes a <BEL> character. Some applications use an alarm to alert the user to particular events. The default setting is to sound the alarm. You can select a key to switch it off and on.
	AUTOSCR	Toggles the setting for autoscrolling of input and output written to the screen. The default setting is to autoscroll; you can select a key to switch it off and on.
	CLOSE	Ends the shell session currently displayed. Close provides the same function as the Quit function key.
	CONTROL	Treats a character on the command line as part of an escape sequence, and does not append a <newline> character to the sequence. For example, if you type `d` on the command line and press the Control function key, the system processes the `d` as the EBCDIC equivalent of the ASCII control sequence <Ctrl-D>.
	ECHO	Toggles the automatic hiding and display of input. If pressed while an application has control over the display of input, the application no longer controls it. If pressed while the application does not control the display of input, the application is given control. See the description of the Hide function key.
	HALFSCR	Scrolls forward half of the currently displayed output.

Table 1. Function Key Settings Available in the OS/390 Shell (continued)

Default Setting	Function/ Subcommand	Description
	HIDE	Toggles the hiding and display of input. If you are using OMVS in ECHO mode, pressing this key overrides the visibility asked for by an application, for the next input only. In NOECHO mode, if the input area is not hidden, pressing this key hides the input area for the next input only. If the input area is already hidden, pressing this key makes the input area visible.
	NO	Deactivates a function key so that it performs no function.
	NOALARM	Performs the same function as Alarm.
	NOAUTO	Performs the same function as Auto.
	NOECHO	Performs the same function as Echo.
	NOHIDE	Performs the same function as Hide.
	NOPFSHOW	Toggles the display of function key settings. The default setting is to display the settings; you can select a key to switch the display off and on.
	PFSHOW	Performs the same function as NoPFShow.
	OPEN	Starts another shell session and automatically switches to it. The session is assigned the next unused session number.
	PREVSESS	Displays the next-lower-numbered session. However, if the lowest-numbered session is currently displayed, the highest-numbered session will be displayed.
	QUIT	Ends the current session, and displays the next-lower-numbered session. However, if the lowest-numbered session is currently displayed, the next-higher-numbered session is displayed. If only one session is active, Quit causes the OMVS command to quit. The workstation returns to TSO/E, and the shell stops processing.
	QUITALL	Ends all active shell sessions. QuitAll causes the OMVS command to quit. The workstation returns to TSO/E. **Note:** If the OMVS interface is running in SHAREAS mode (shared address space) and you quit all sessions (QuitAll or Quit if there is just one session), the shell process ends immediately.

The Escape Character

ESC=¢

An escape sequence produces an EBCDIC version of the ASCII control sequence. (For example, the OS/390 UNIX <EscChar-D> corresponds to the ASCII <Ctrl-D>.) If you do not use a Control function key to enter escape sequences, you will need to use an escape character. When you

type an escape character followed by a second character and press Enter, the second character is converted into a different character before it is passed to the shell.

The default escape character depends on the character conversion table specified with the CONVERT keyword. For more information, see the **OMVS** command description in *OS/390 UNIX System Services Command Reference*.

There can be up to eight escape characters defined and displayed on the screen; you can use any one of them as an escape character. For example, three are displayed here:

```
ESC=¢`%
```

In this book, the notation *EscChar* coupled with another letter (for example, <EscChar-D>) indicates an escape sequence.

For more information about escape sequences, see "Typing Escape Sequences in the Shell" on page 22, which follows.

Entering a Shell Command

You type shell commands on the shell command line (===>) and press <Enter> to pass them to the shell.

Customizing the Variant Characters on Your Keyboard

If the shell is using a locale generated with code pages IBM-104 IBM-1027, or IBM-939, an application programmer needs to be concerned about "variant" characters in the POSIX portable character set whose encoding may vary from other EBCDIC code pages. For example, the encodings for the square brackets do not match on code pages IBM-037 and IBM-1047:

Left square bracket: [(X'AD' on IBM-1047)

Right square bracket:] (X'BD' on IBM-1047)

You may want to customize the encodings for those keys on your keyboard. See "Appendix E. Code Page Conversion when the Shell and MVS Have Different Locales" on page 345 for more information on this topic.

Entering a Long Shell Command

If you are typing a long command that will not fit on the command line, you can use the \ (backslash) continuation character at the end of the first line. When you then press <Enter>, the command line is cleared so that you can continue typing. The line you typed prior to the backslash is displayed in the output area, and beneath it the shell prompt changes to > to indicate that you are continuing a command. For example:

```
$ cat /usr/macneil/uts/mydir/mydata\
>









===> /applprog/dbprog/dbget.c
                                                          RUNNING
```

While the shell is processing your command, it displays the RUNNING status indicator.

Where's the Command Output? If your output has not yet been displayed when the status changes to INPUT, press the Refresh function key to see the output.

Entering a Shell Command from TSO/E

You can use the OSHELL REXX exec to run an OS/390 shell command from the TSO/E READY prompt and display the output to your terminal. The syntax is:

oshell *shell_command*

With this exec, do not use an & to run a shell command in the background. See "OSHELL: Running a Shell Command from TSO/E READY" on page 166 for more information.

Interrupting a Shell Command

If you want to interrupt a command and stop it from completing, type <EscChar-C> or type c and press the Control function key (if you have a function key customized to perform the Control function; see "Determining Function Key Settings and the Escape Character" on page 17).

Typing Escape Sequences in the Shell

An escape sequence produces an EBCDIC version of the ASCII control sequence. (For example, the OS/390 UNIX <EscChar-D> corresponds to the ASCII <Ctrl-D>.) You can use escape sequences to type:

- Portable characters not included on your keyboard; see Appendix F for these sequences.
- Control characters that are normally available on ASCII workstations, but not EBCDIC ones; see Appendix F for these sequences.

In this book, the notation *EscChar* coupled with another letter (for example, <EscChar-D>) indicates an escape sequence, corresponding to an ASCII control sequence. You can type an escape sequence in either of these ways:

- Type a letter on the command line and press the Control function key if you have one defined. The Control function key treats the character on the command line as if it were preceded by an escape character, and it does not append a <newline> character.

For example, to exit the shell, you type `d` on the command line and press the Control function key.

To use a Control function key, you must customize the **OMVS** command with a key setting for that function.

- Type an escape character sequence, beginning with one of the escape characters. After you type the two characters in sequence and press <Enter>, the system translates the two characters into a third character. For information on how to customize your keyboard for typing an escape sequence, see “Keyboard Remapping”.

Suppressing the Newline Character

Whenever you press <Enter>, a <newline> character is automatically appended to the characters you typed. For certain UNIX applications, you may want to suppress the automatic <newline> character appended when you press <Enter>.

If you use the Control function key to input an escape sequence, no <newline> character is appended. However, if you use an escape character to input an escape sequence, a <newline> character is appended to the sequence. To suppress the <newline> character, add an escape character at the end of the input and press <Enter>.

For example, in the shell, the two-character EBCDIC sequence <EscChar-D> is the equivalent of the ASCII control sequence <Ctrl-D>. To enter only an <EscChar-D> with no final <newline>, type the string <EscChar-D-EscChar> on the command line, and press <Enter>; an example is shown in Figure 10.

```
===>  ¢d¢
                                                                  INPUT <3>
ESC=¢  1=Help    2=SubCmd   3=HlpRetrn   4=Top      5=Bottom   6=TSO
       7=BackScr 8=Scroll   9=NextSess  10=Refresh 11=FwdRetr 12=Retrieve
```

Figure 10. Typing an Escape Sequence

Keyboard Remapping

With most terminal emulators, you can use the keyboard remapping function to define one key to generate a multikey sequence. For example, you could define the <D> key so that it generates <EscChar-D-EscChar-Enter> when the <Ctrl> key is pressed in sequence with it. Thus, the sequence <Ctrl-D> acts like the ASCII sequence <Ctrl-D>.

Determining Your Session Status

To find out the status of your session, look in the lower right-hand corner of the screen. A status indicator and shell session number (if more than one session has been started) are displayed. The session number identifies the session displayed on your screen.

```
                    INPUT <1>
 5=Bottom    6=TSO
11=FwdRetr  12=Retrieve
```

The status indicators are:

INPUT Indicates that the shell is ready for input and will not send any more output to the display screen. If your output has not yet been displayed when the status changes to `INPUT`, press the Refresh function key to see more output.

RUNNING
Indicates that the workstation and shell are being polled, or that an application program is running. After the polling completes, the indicator changes.

MORE...
Indicates the screen is full of output and there is more output waiting to be displayed. To scroll the screen, do one of the following:

- Press the <Clear> key (or combination of keys, depending on your keyboard)
- Press the Scroll, HalfScr, or Bottom function key.

INPUT HIDDEN
Indicates that you have pressed a function key that will turn off the display of any input that you type. Typically, this function is used for typing in secure data. Once you press <Enter>, any further input is displayed.

HIDDEN is a short form of INPUT HIDDEN, used when it is combined with other status indicators, such as:

- HIDDEN/MORE
- HIDDEN/INPUT
- HIDDEN/NOT ACCEPTED
- HIDDEN/NOTACC/MORE
- HIDDEN/NOTACC/INPUT

NOT ACCEPTED
Indicates that the application or shell is hung and not accepting any input you enter. Try using a subcommand to interrupt the application.

NOTACC is a short form of NOT ACCEPTED, used when it is combined with other status indicators, such as:

- HIDDEN/NOTACC/MORE
- HIDDEN/NOTACC/INPUT

NOT ACCEPTED/MORE...
Indicates that the application is not accepting any input you enter and that there is more output waiting to be displayed. Scroll the screen to clear it before trying to reenter the input. To scroll the screen, do one of the following:

- Press the <Clear> key (or combination of keys, depending on your keyboard)
- Press the Scroll, HalfScr, or Bottom function key.

SUBCOMMAND
Indicates that you are working in subcommand mode.

Scrolling through Output

You can scroll output forward and backward using:
- Function Keys
- Cursor scrolling
- Scrolling subcommands

Using Function Keys or Subcommands

There are five scrolling function keys that you can use during a shell session and in subcommand mode:

- BackScr
- Bottom
- HalfScr
- Scroll
- Top

These are discussed in Table 1 on page 17.

You can use the TOP, BOTTOM, SCROLL, BACKSCR, and HALFSCR subcommands to scroll output. They produce the same results as their corresponding function keys (see Table 1 on page 17).

Using Cursor Scrolling

Cursor scrolling gives you better control over the scrolling action. You place your cursor on a line in the output and then press one of these function keys or type the corresponding subcommand:

BackScr
Positions the output line containing the cursor at the bottom of the output displayed on the screen.

If the output partially fills the screen and the cursor is positioned below the last line of output, the empty line with the cursor is displayed at the bottom of the screen.

HalfScr
Positions the output line containing the cursor near the center of the output displayed on the screen. This is similar to a partial scroll forward or backward.

If the output partially fills the screen and the cursor is positioned below the last line of output, the empty line with the cursor is displayed near the center of the screen.

Scroll Positions the output line containing the cursor at the top of the output displayed on the screen.

If the output partially fills the screen and the cursor is positioned below the last line of output, the last line of output is displayed at the top of the screen.

Running a Subcommand

A subcommand is a command that is passed to the **OMVS** command processor (instead of to the shell). Most subcommands are used to control, or temporarily change, the OMVS interface. You can issue a subcommand in two ways:

- Type the subcommand on the shell command line and press the Subcommand function key, if you have one defined.
- Switch to subcommand mode and enter the subcommand.

The names of the subcommands match the names of the functions listed in Table 1 on page 17. Some subcommands have aliases; for information on the subcommands and their aliases, see the **OMVS** command description in *OS/390 UNIX System Services Command Reference* .

You can enter the subcommands in uppercase, lowercase, or mixed-case letters.

Switching to Subcommand Mode

Instead of using the Subcommand function key to run a subcommand, you can switch to subcommand mode to enter it. To switch to subcommand mode, press the SubCmd function key when the shell command line is empty. In subcommand mode, the screen appears as it did in the shell, except in the lower right-hand corner the status displayed is `SUBCOMMAND`. Existing output from the shell is displayed at the top of the screen, and any new output is displayed as it is available.

When you switch to subcommand mode, the command prompt changes to `OMVS Subcommand ==>`.

Using Multiple Sessions

You can run more than one shell session concurrently. When you have more than one session active, the sessions are numbered and the identifying number for a session is displayed next to the status indicator.

Starting Sessions

To start additional sessions, you can:

- Use the SESSIONS keyword on the **OMVS** command to specify the number of sessions you want started automatically when you log into the shell. By writing a small REXX program or CLIST, you can customize your invocation of the **OMVS** command so that every time you log into the shell multiple sessions are started.
- Use the OPEN subcommand during a session. This starts another shell session and automatically switches to it. The session is automatically assigned the next unused session number.

Switching between Sessions

You can use function keys or subcommands to switch between sessions:

- NextSess is the default setting for <F9>. If you wish, you can customize an additional key for the PrevSess setting. See Table 1 on page 17 for a discussion of these functions.
- The NEXTSESS and PREVSESS subcommands perform the same as the function keys.

Customizing the OMVS interface

You can select the keywords you want to use when you enter the TSO/E **OMVS** command:

```
ALARM | NOALARM
AUTOSCROLL | NOAUTOSCROLL
CONVERT()
```

```
DBCS | NODBCS
DEBUG()
ECHO | NOECHO
ENDPASSTHROUGH(ATTN | CLEAR | CLEARPARTITION | ENTER | NO | PA1 | PA3 | PF1
 ... PF24 | SEL)
ESCAPE()
HIDE | NOHIDE
LINES()
PF()
PFSHOW | NOPFSHOW
RUNOPTS()
SESSIONS()
SHAREAS | NOSHAREAS
WRAPDEBUG()
```

An Example of Customizing the OMVS Command

Say you want to invoke the **OMVS** command to:

- Set function key 1 as the Control function key
- Start three sessions
- Not use a shared address space

You would enter:

```
omvs pf1(control) sessions(3) noshareas
```

By writing a small REXX program or CLIST, you can customize your selection of keywords on the TSO/E **OMVS** command. If you intend to use these settings every time you enter the command, you could:

1. Write a REXX program that runs the **OMVS** command with the customized keywords. For example, here is a REXX program called MYOMVS:

   ```
   /* REXX */
   P = PROMPT("ON");                 /* Don't suppress prompting */
   "omvs pf1(control) sessions(3) NOSHAREAS";
   X = PROMPT(P);                    /* Restore original prompting state */
   Return;
   ```

   The use of the REXX function PROMPT() is required to prevent prompts from being suppressed. Otherwise, TSO/E commands cannot prompt you for additional information when the commands are issued during a shell session.
2. Install the exec in a data set that is part of either the SYSPROC or SYSEXEC concatenation.
3. When you log on to TSO/E, at the `READY` prompt you enter `MYOMVS` and the exec calls MYOMVS, your customized **OMVS** command. Your changes override the default settings.

For more discussion of the syntax of the **OMVS** command and its customizable keywords, see *OS/390 UNIX System Services Command Reference*.

The Alarm Setting (ALARM|NOALARM)

Some applications sound an alarm to alert the user to particular events. To change the default alarm setting (which allows it to sound), use the NOALARM keyword.

Autoscrolling (AUTOSCROLL|NOAUTOSCROLL)

Automatic scrolling of input and output written to the screen is the default. Specify NOAUTOSCROLL to prevent this.

If an application writes a <form-feed> character with no following data to a terminal and OMVS is in AUTOSCROLL mode, the screen is cleared.

The Character Conversion Table (CONVERT)

There are both APL and non-APL character conversion tables. The IBM-supplied default is a null conversion table, but the system programmer can select a different default for the **OMVS** command to use. If you do not want to use the default table, you can specify a table name with the CONVERT keyword. See *OS/390 UNIX System Services Command Reference* for more details.

To access data in the hierarchical file system (HFS), use a terminal that is operating in the same code page as the HFS. In other words, if you have a 3270 terminal using a French code page, you cannot access HFS data encoded in a German code page when you are using the OMVS-provided character conversion tables. However, you could provide your own OMVS conversion tables to convert between the French and German code pages.

Doublebyte Character Set Support (DBCS|NODBCS)

By default, the **OMVS** command supports the use of a doublebyte character set (DBCS). If your terminal does not support DBCS, this default has no effect. To prevent DBCS processing on a DBCS terminal, specify the NODBCS keyword.

You may want to use the NODBCS option if you have a DBCS terminal but do not want the overhead associated with using the **OMVS** command with DBCS support.

Debugging for the OMVS Command (DEBUG)

Change this default setting from NO only if IBM asks you to do so. To control the collection and output of debugging information, change the DEBUG keyword as directed.

Giving an Application Control of the Command Line (ECHO|NOECHO)

You can use the ECHO option to allow an application to control the input area's visibility. When ECHO is specified, OMVS hides or displays the input area based on the application's setting of the ECHO bit in the termios structure. If the bit is off, the command line is hidden, except in subcommand mode. If the bit is on, the command line is visible. The default is NOECHO, which does not allow the application to control the visibility of the input area.

Ending 3270 Passthrough Mode (ENDPASSTHROUGH)

Applications running from the shell can switch to TSO3270 passthrough mode, which lets an application invoke TSO/E functions. For application development purposes, you can specify a key that will end TSO/3270 passthrough mode and force OMVS to return to the shell session.

Because this key is used only during application development, the default is ENDPASSTHROUGH(NO).

For more information about TSO/3270 passthrough mode, see *OS/390 UNIX System Services Programming Tools*.

The Escape Character (ESCAPE)

If you do not use a Control function key to escape a character, you can type a two-character escape sequence instead. (For an explanation of escape characters, see "Typing Escape Sequences in the Shell" on page 22.)

To change the default escape character, or have more than one escape character, type escape characters after the ESCAPE keyword. You can type up to eight characters, enclosed in single quotes with no space between them. For example:

```
OMVS ESCAPE('`¢')
```

When specifying escape characters:

- Select characters that are not in the POSIX portable character set. See "The POSIX Portable Character Set" on page 346 to see the contents of the POSIX portable character set.
- Select singlebyte characters, even if you are using a doublebyte character set.

The escape characters specified with the **OMVS** command completely override those in the character conversion table being used. However, if no escape characters are specified with the **OMVS** command, the system uses those in the conversion table.

Controlling the Size of the Output Scroll Buffer (LINES)

You can override the default size of the output scroll buffer; the default is roughly four screenfuls. With the LINES keyword, you can specify the size of the buffer; the range is 25 to 3000 lines.

Note: Using a large output scroll buffer increases the amount of storage that the **OMVS** command requires; it also causes additional overhead, impacting performance.

Function Key Settings (PFn)

To customize any of the default function key settings, type your selection in the parentheses after the function key name. For example:

```
OMVS PF1(CONTROL)
```

makes function key 1 the Control key, which you use to type an escape sequence such as <Ctrl-D> (first you type `d` on the command line, and then you press the function key).

Displaying the Function Key Settings (PFSHOW|NOPFSHOW)

To turn off the display of function key settings, specify the NOPFSHOW keyword on the **OMVS** command.

Specifying Language Environment Runtime Options (RUNOPTS)

To run the TSO/E **OMVS** command with LE runtime options, specify the RUNOPTS keyword. For example:

```
OMVS RUNOPTS('RTLS(ON) RPTOPTS(ON)')
```

will run the **OMVS** Command with RTLS(ON) and will print out an options report. See *OS/390 Language Environment Programming Reference* for a list of runtime options.

Note: The use of inappropriate LE runtime options, such as TRAP(OFF) or POSIX(OFF), may cause the **OMVS** command to fail. The intended use of the RUNOPTS keyword is to specify the RTLS, LIBRARY, and VERSION LE runtime options.

Any valid run-time options specified by RUNOPTS normally get passed along to the shell.

Multiple Sessions (SESSIONS)

If you want more than one session started when you invoke the **OMVS** command, use the SESSIONS keyword. The suggested maximum number of sessions is three or four. If you try to start too many sessions (the limit depends on the size of your TSO/E address space), your TSO/E user ID runs out of storage and various unpredictable errors may occur. You may have to log off your TSO/E user ID before you can continue.

The Shared TSO/E Address Space (SHAREAS|NOSHAREAS)

Having the **OMVS** command and the shell run in the same (shared) TSO/E address space saves one address space per user and simplifies transaction accounting, as managed by the operating system. The shell shares the address space (SHAREAS) by default, unless the shell is a SETUID or SETGID program and the owning UID or GID is not the same as the current owner.

If you specify NOSHAREAS, the shell may keep running even after the QUIT subcommand has been entered; in most cases, it will not.

For more information about shared address space, see Chapter 12.

Controlling Data Recorded in the Debug Data Set (WRAPDEBUG)

Use the WRAPDEBUG keyword to specify how many lines of debug data OMVS writes out before wrapping around to the top of the debug data set.

Performing TSO/E Work or ISPF Work after Invoking the Shell

After you have invoked the shell, you can:

- Enter a TSO/E command from the command line
- Switch temporarily to TSO/E command mode
- Return to ISPF or the TSO/E `READY` prompt

Entering a TSO/E Command from the OS/390 Shell

You can enter a TSO/E command from the shell in either of these ways:

- Type the **tso** shell command before the TSO/E command. For example:

  ```
  tso "oput source.c(hello) '/u/ehk/source/hello.c'"
  ```

  Note that the **oput** command is quoted so that the shell does not process it. If you are copying a file, specify the *-t* option to copy a file to your current directory. For more information about the **tso** command and its options, see *OS/390 UNIX System Services Command Reference*.

- Type the command on the shell command line and press the TSO function key.

  When the TSO/E command completes, typically `***` is displayed on the screen. To return to the shell and resume working at the shell command line, press <Enter> or <Clear>.

You can use the **man** command to view manual descriptions of TSO/E commands. To do this, you must prefix all commands with **tso**. For example, to view a description of the MOUNT command, you would enter:

```
man tsomount
```

For complete information about the **man** command, see*OS/390 UNIX System Services Command Reference*.

Command Not Found? If you type a TSO/E command from the shell and press <Enter> instead of the TSO function key, you may receive a message that the command is "`not found`". Because you did not press the TSO function key, the shell attempted to process the command as a shell command. (You can use the Retrieve function key to redisplay the command.)

Switching to TSO/E Command Mode

There are two contexts for switching to TSO/E command mode:

- You are in the OS/390 shell. You want to run TSO/E commands without shutting down any processes that may be running and without exiting the shell completely.
- You are in subcommand mode and want to run TSO/E commands.

You can switch to TSO/E command mode to run TSO/E commands (such as **OPUT** or **OGET**). When the command line is empty, press the TSO function key. Any shell scripts or processes that were running when you pressed the function key continue to run.

After you are in TSO/E command mode, the screen is in line mode and no function keys are active or displayed. A special prompt (not the typical TSO/E `READY` prompt) is issued; the prompt is:

```
OMVS - Enter a TSO/E command, or press PA1 to return to the shell.
```

When you complete your work in TSO/E command mode, press <PA1> to return to wherever you were before you entered TSO/E. You can resume your work in the shell or return to subcommand mode.

ftp or telnet from TSO

There are no **ftp** or **telnet** shell commands available in the shell. However, when you use the **OMVS** command to login to the shell, you can switch to TSO and issue the **ftp** or **telnet** command from there, with the following restrictions:

- You can use **ftp** to access MVS sequential and partitioned data sets. If the TCP/IP OpenEdition Extended Applications feature is installed, you can ftp directly into the HFS.
- When you **telnet** to a remote MVS host and then access a shell, you can work in line mode only (for example, you cannot use **vi**).

Exiting the Shell

There are four situations when you might want to exit the shell:

- **To leave the shell temporarily and switch to TSO/E command mode:** Press the TSO function key. You can do this any time during a session, regardless of whether you are currently running a command or script. See "Performing TSO/E Work or ISPF Work after Invoking the Shell" on page 30 for details.

  If you switch to TSO/E command mode, the shell and any shell commands continue running until they attempt to read from the terminal or until the terminal output buffer is full; if either of these situations occurs, the commands are suspended until you return to the shell.

- **To exit the shell when a foreground process has completed:** Type **exit** or <EscChar-D>. Scroll past all the output data (or use an autoscroll function key if you have customized a function key to do that), and exit.

  Note: The <EscChar-D> sequence does not work if you have entered `set -o ignoreeof` in the shell. See the description of **set** in *OS/390 UNIX System Services Command Reference*.

  If you are using the shell option **set +m** or its equivalent **set +o monitor** to have background jobs run in the same process group as the shell, use the **nohup** command to run a script or program that will continue running after you log out.

  If you were in ISPF when you entered the shell, you are returned to ISPF; if you were in TSO READY mode, you are returned to TSO/E `READY`.
- **To exit the shell when a background job is running,** press the SubCmd function key and then enter the QUIT subcommand.

  Note: If your OMVS interface is running in SHAREAS mode (shared address space) and you quit all sessions (QUITALL subcommand or QUIT for the only session), the shell process ends immediately.

  If you were in ISPF when you entered the shell, you are returned to ISPF; if you were in TSO/E READY mode, you are returned to TSO `READY`.

  By default in the shell (the **set -m** option), a background job runs in a different process group from the shell, and the job keeps running after you exit the shell. To have background jobs run in the same process group as the shell, use the **set +m** command or its equivalent, **set +o monitor**.
- **If your application is in a loop,** try using <EscChar-C> or <EscChar-V> to interrupt it. If this does not work, press the SubCmd function key to leave the shell. Then type `quit` and press <Enter>. This causes the **OMVS** command to quit abruptly. The workstation returns to TSO/E and the shell stops processing. For more information on using escape sequences such as <EscChar-C>, see "Typing Escape Sequences in the Shell" on page 22.

Getting Rid of a Hung Application

If your application hangs, try the following procedure to kill it:

1. On the command line, enter <EscChar-V> (or <EscChar-C>). When this is successful, the shell prompt is displayed.
2. If step 1 does not work, enter the OPEN or NEXTSESS subcommand to start or switch to a second shell session. In the second shell session, determine the process identifier (PID) of the hung application by entering `ps -ef`.

   Then enter `kill -s KILL` *nnnnnn*, where *nnnnnn* is the PID obtained from the **ps -ef** command. After the **kill** command completes, you can return to the first session using the NEXTSESS or PREVSESS subcommand.
3. If step 2 does not work, enter the QUIT subcommand, or QUITALL if more than one session is active. This should free your TSO/E terminal, and you can then enter the **OMVS** command to start another session. The application may still be hung; if so, you need to use the **kill** command.
4. If step 3 does not work, ask the operator to cancel your TSO/E user ID, using the CANCEL command. The operator may also need to use the FORCE command.
5. If step 4 does not work, try a VTAM logoff (using the <SYSREQ> key), and wait long enough for MVS to end your session before you try to log on again.

Using a Doublebyte Character Set (DBCS)

If you want to display or enter doublebyte data, you must:

- Work at a terminal that is configured to generate data in code page IBM-939 and follow the procedures for the terminal emulator being used, if any.
- Specify special LOGMODEs to access TSO/E and VTAM support for DBCS. Typically the system programmer will have set these up and provided you with instructions.
- Run the TSO/E PROFILE PLANGUAGE(JPN) command, if required, to receive Japanese-language messages from the OMVS interface to the shell. Do not change your PROFILE PLANGUAGE when temporarily switched to TSO/E from the shell. After you have invoked the shell, OMVS will not change the language of the messages it issues until you exit the shell and return to TSO/E, change your PROFILE PLANGUAGE, and re-invoke OMVS.
- Use the null translate table (the default) for character conversion. You do not need to specify the CONVERT keyword on the **OCOPY**, **OGET**, **OGETX**, **OPUT**, and **OPUTX** commands.
- Access the shell using the **OMVS** command with the DBCS keyword, the default setting.
- Define a singlebyte escape character for typing an escape sequence, if you do not use the default ¢.

The shell utilities (for example, **grep** and **ed**) work with DBCS data in the file system and can be used to create DBCS data in the file system.

Singlebyte Restrictions

When working with a doublebyte character set, you must use singlebyte characters in these situations:

- Singlebyte characters for filenames. DBCS characters in filenames will be treated as SBCS characters.
- Singlebyte characters for command-line options
- Singlebyte characters for command-line arguments
- Singlebyte characters for delimiters such as a slash, curly brackets, parenthesis, and so on
- For user-defined environment variables, only SBCS for the names, and SBCS or DBCS for the values
- For the shell environment variables, only **IFS**, **PS1**, and **PS2** support DBCS values

The system programmer should use:

- Only singlebyte characters for user names and passwords
- Only singlebyte characters for device, group, and terminal names

Chapter 3. The Asynchronous Terminal Interface to the Shells

For people who have worked with UNIX systems, the asynchronous terminal interface is quite familiar. You use the asynchronous terminal interface if you access the OS/390 shells with one of these methods:

- **rlogin**
- **telnet**
- **rlogin** or **telnet** via the Communications Server
- Communications Server login from a serially attached terminal

ASCII-EBCDIC Translation

When you use **rlogin**, **telnet**, or Communications Server to access the shell, the data you enter is translated from ASCII (ISO8859-1) to EBCDIC (IBM-1047) before the shell processes it. To change code pages for the current session, use the **chcp** command. To automatically change code pages after you login, see “Changing the Locale in the Shell” on page 45 for the OS/390 shell, or “Changing the Locale in the Shell” on page 59 for the tcsh shell.

For a complete list of the singlebyte and doublebyte ASCII and EBCDIC code pages that you can specify, see the *OS/390 C/C++ Programming Guide*.

Using rlogin to Access the Shell

When the **inetd** daemon is set up and active, you can **rlogin** to a shell from a workstation that has **rlogin** client support and is connected via TCP/IP or Communications Server to the MVS system. To login, use the **rlogin** command syntax supported at your site.

To improve performance when you **rlogin** into a shell, you can use shared address space; for more information, see Chapter 12.

Note: If you are writing or porting an **rlogin** command to rlogin into a shell, the shell interface to **rlogin** consists of the FOMTLINP and FOMTLOUT modules, documented in *OS/390 UNIX System Services Planning*.

Using telnet to Access the Shell

You can **telnet** to the shell from a workstation that is connected via TCP/IP or Communications Server to the MVS system. Use the **telnet** command syntax supported at your site.

Using Communications Server login to Access the Shell

If you are working at a terminal that is serially attached to the Communications Server, you can login directly to the shell.

1. Specify the host you want to login to. You receive a message confirming that you are connecting to the host.
2. At the prompts, enter your user ID and password.

The Shell Session

Once your login completes, you see your normal shell prompt (for example, $ or >). This is a UNIX interface, not the 3270-type interface that is displayed by the OMVS command. By default, the terminal interface is in line mode (also known as canonical mode), which means that each time you type a command at the prompt, you need to press Enter to process the command. Some utilities switch the terminal interface to raw mode. When you use a raw mode utility (such as **vi** or **talk**), or when command line editing is enabled in the shell, each keystroke is transmitted; you do not need to press <Enter>.

When you are in a shell session, you can:

- Run all shell commands and utilities.
- Run any application from the hierarchical file system (HFS).
- Use the **vi editor** and other full-screen applications such as **talk** and **more**.

In the OS/390 UNIX environment, the asynchronous terminal interface session has some differences from an OMVS session:

1. You cannot switch to TSO/E. However, you can use the **tso** shell command to run a TSO/E command from your session.
2. You cannot use the ISPF editor. (This includes the **oedit** and TSO/E OEDIT commands, which invoke ISPF File Edit.)

Entering a Shell Command

You type shell commands and press <Enter> to pass them to the shell.

If you are typing a long command that will not fit on one line, you can use the \ (backslash) continuation character at the end of the first line. When you then press <Enter>, the line is cleared so that you can continue typing. The line you typed prior to the backslash is displayed in the output area, and beneath it the shell prompt changes to > (? in tcsh) to indicate that you are continuing a command.

Interrupting a Shell Command

If you want to interrupt a command and stop it from completing, type <Ctrl-C>. The command stops executing and the system displays the shell prompt. You can now enter another command.

Using Multiple Sessions

With **rlogin**, **telnet**, or Communications Server, you can login to a shell more than once, using the same user ID and password. You can also be logged in to a shell using the OMVS 3270 interface and the asynchronous terminal interface at the same time, using the same user ID and password.

Using a Doublebyte Character Set (DBCS)

If you want to display or enter doublebyte data:

- You must work at a terminal that is configured to generate data in code page IBM-939 and follow the procedures for the terminal emulator being used, if any.
- Customize your locale and use the **chcp** command to specify the ASCII and EBCDIC code pages you are using.

– For information on how to customize your locale and configure your setup files, see “Changing the Locale in the Shell” on page 45 for the OS/390 shell, or “Changing the Locale in the Shell” on page 59 for the tcsh shell.

When you are working with a doublebyte character set, there are some restrictions. See “Singlebyte Restrictions” on page 33 for more information.

Standard Shell Escape Characters

The following are some of the standard shell escape characters:

- <Ctrl-C> — Program interruption
- <Ctrl-D> — End of file
- <Ctrl-V> — Quit Program
- <Ctrl-Z> — Suspend Program

Chapter 4. Customizing the OS/390 Shell

If you are interested in working with the OS/390 shell, read this chapter and:

- “Chapter 6. Working with OS/390 Shell Commands” on page 67
- “Chapter 8. Writing OS/390 Shell Scripts” on page 121

You can personalize your use of the OS/390 shell. This chapter covers these topics:

- Creating or modifying your **.profile** file
- Understanding shell variables
- Customizing your shell environment with the **ENV** variable
- Customizing the search path for commands with the **PATH** variable
- Improving the performance of shell scripts
- Changing the locale
- Customizing the language of messages
- Setting the time zone
- Building a STEPLIB environment
- Setting options for a shell session

Customizing Your .profile

When you start the OS/390 shell, it uses information in three files to determine your particular needs or preferences as a user. The files are accessed in this order:

1. **/etc/profile**
2. **$HOME/.profile**
3. The file that the ENV variable specifies

Settings established in a file accessed earlier can be overwritten by the settings in a file accessed later.

The **/etc/profile** file provides a default systemwide user environment. The system programmer may modify the variables in this file to reflect local needs (for example, the time zone or the language). If you do not have an individual user profile, the values in the **/etc/profile** are used during your shell session.

The **$HOME/.profile** file (where **$HOME** is a variable for the home directory for your individual user ID) is an individual user profile. Any values in the **.profile** file in your home directory that differ with those in the **/etc/profile** file override them during your shell session. OS/390 provides a sample individual user profile. Your administrator may set up such a file for you, or you may create your own.

Typically, your **.profile** might contain the following:

```
export ENV=$HOME/.setup  #set and export ENV variable
export PATH=$PATH:$HOME:  #set and export PATH variable
export EDITOR=ed          #set and export EDITOR variable
export PS1='$LOGNAME':'$PWD':' >'
```

Figure 11. A Sample .profile

If the value on the right-hand side of the **=** sign does not contain spaces, tab characters, or other special characters, you can leave out the single quotes.

Each of the lines begins with an **export** command. For the Korn shell, this sets the variable and also specifies that whenever a subshell is created, these variables should be exported to it. You can also set a variable on one line and export it on another, as shown here:

```
ENV=$HOME/.setup
export ENV
```

If portability to a Bourne shell is a consideration, use the two-line syntax. See "Exporting Variables" on page 124 for more information about exporting variables.

export ENV=$HOME/.setup
: Identifies the **.setup** file in your home directory as your login script (also known as a setup script or environment file) and specifies that whenever a shell is created, the **ENV** variable should be exported to it. See "Customizing Your Shell Environment: The ENV Variable" on page 42 for more information about a login script.

export PATH=$PATH:$HOME:
: Identifies the search path to be used when locating a file or directory, and specifies whenever a subshell is created, the **PATH** variable should be exported to it. Here, the system first searches the path identified in the **PATH** variable in **/etc/profile**, the system profile; then the system searches your home directory; finally, the system searches your current working directory. A leading or trailing colon, or two colons in a row, represents the current working directory. To avoid confusion, this is often expressed as:

  ```
  export PATH=$PATH:$HOME:.
  ```

  This PATH setting and the one in the example are equivalent. See "Customizing the Search Path for Commands: The PATH Variable" on page 43 for more information.

export PS1='$LOGNAME:$PWD: >'
: Identifies the shell prompt that indicates when the shell is ready for input, and specifies whenever a subshell is created, the **PS1** variable should be exported to it. Here the prompt (default is $) has been customized to show your login name and working directory. For example, for user ID TURBO working in the home directory, the prompt would display as:

  ```
  turbo:/u/turbo: >
  ```

  When TURBO changes directories, the prompt changes to indicate the working directory.

export EDITOR=ed
: Identifies **ed** as the default editor used by some of the utilities, such as **mailx**, and specifies whenever a subshell is created, the **EDITOR** variable should be exported to it.

If you create a subshell with the command **sh –L**, the shell starts and reads and executes your profile file. Note that the letter **L** must be in uppercase. The shell looks for **.profile** in the **$HOME** directory. If it is not found, the shell looks in the working directory; therefore, make sure that you are working in the right directory when you enter this command.

Quoting Variable Values

When you have blanks in a variable value, you need to enclose it in quotes. The quotes tell the shell to treat blanks as literals and not delimiters. Single quotes are more "serious" about this than are double quotes:

- Single quotes preserve the meaning of (that is, treat literally) all characters.
- Double quotes still allow certain characters ($, ` (backquote), and \ (backslash)) to be expanded. This is important if you want variable expansion. For example, see how the $ is handled here:

  ```
  export HOMEMSG="Using $HOME as Home Directory"
  ```

  If your home directory were set to **/u/user**, the following:

  ```
  echo $HOMEMSG
  ```

  would display:

  ```
  Using /u/user as home directory
  ```

  If, instead, you enclosed the variable value in single quotes, like this:

  ```
  export HOMEMSG='Using $HOME as home directory'
  ```

  the following:

  ```
  echo $HOMEMSG
  ```

  would display:

  ```
  Using $HOME as home directory
  ```

  As you can see, the $ is not expanded.

Changing Variable Values Dynamically

You can also change any of these values for the duration of your session (or until you change them again). You enter the name of the environment variable and equate it to a new value. For example:

```
PS1='+>'
```

changes the command prompt string to +>.

Understanding Shell Variables

You can display the shell's variables and their values by entering this command:

```
set
```

You may see many variables that you don't recognize. These are *built-in*, or *predefined*, variables that are set up with default values when you start the shell.

You can customize the built-in variables by setting their value in your **.profile**. Only the variables **IFS**, **PS1**, and **PS2** support doublebyte characters for the values.

Only the shell variables that are exported are available to shell scripts and commands invoked from the shell. Environment variables are a subset of shell variables that have been exported.

You can display the environment variables and their values by entering either of these commands:

```
env
printenv
```

You can display the value of a single variable with the **echo** command, the **print** command, or the **printenv** command. For example, any of these commands

```
echo $HOME

print $HOME

printenv $HOME
```

displays the current value of the **HOME** variable.

In general, **echo** displays the current values of all its arguments, after any shell processing has taken place. For example, consider:

```
echo *.doc
```

The shell first expands the wildcard character *****. This produces the names of every file in the working directory that has the suffix **.doc**. So the output of **echo** is a list of all such files. And if there are no filenames ending in **.doc**, the command output is just `*.doc`.

For more information about shell variables,

- Built-in variables are listed in a table in the **sh** command description in *OS/390 UNIX System Services Command Reference*.
- There is an appendix that lists shell variables in *OS/390 UNIX System Services Command Reference*.

Customizing Your Shell Environment: The ENV Variable

So far, we have discussed customization that is set up inside your **.profile** file. However, the shell reads your profile file only when you log into the shell or when you enter the **sh** command **–L** option.

To always have a customized shell session, you need to have a special shell script that sets up the environment started each time you start the shell; this is called a *login script* (also known as an environment file, or startup script). You specify the name of this script in the **ENV** variable in your **.profile** file.

When you start the shell, the shell looks for an environment variable named **ENV**. You can use the **ENV** variable to point to a login script that sets things up in the same way that the profile file does.

For example, you might put all your alias definitions and other setup instructions into a file called **.setup** in your home directory. You want these instructions run when your shell starts after you login and whenever you explicitly create the shell during a session (for example, as a child shell to run a shell script). To make sure **ENV** is set up when you login or when you execute a shell, specify **export ENV** in your **.profile** file. For example:

```
export ENV=$HOME/.setup
```

You may find it useful to put all your aliases in the login script that **ENV** points to, instead of in your **.profile** file. However, you should keep exported variable assignments in your profile, so that they are run only once.

Customizing the Search Path for Commands: The PATH Variable

Command interpreters usually have to *search* for a file that contains the command you want to run. When using the shell, you tell the shell where to search for a command. Essentially, the shell uses a list of directories in which commands may be found. This list is specified in your **PATH** variable in your **.profile** file. The list could be called your *search path,* because it tells the shell where you want to search.

You can set up a search path with a command of the form:

```
PATH='dir:dir:...'
```

For example, you might enter:

```
PATH='/bin:/usr/bin:/usr/etc:/usr/macneil/bin:/usr/games:/usr'
```

The shell then searches the directories in the following order, when looking for commands or shell scripts:

1. **/bin**
2. **/usr/bin**
3. **/usr/etc**
4. **/usr/macneil/bin**
5. **/usr/games**
6. **/usr**

As soon as the shell finds a file with an appropriate name, it runs that file.

Because the shell runs a command as soon as it finds a file with an appropriate name, pay close attention to the order in which you list directory names in your search path. For example, the previous search path specifies the **/bin** directory (where OS/390 shell commands are stored) before the **/etc** directory.

If you set up your **PATH** incorrectly, you could get the wrong command. You should always search the shell commands directory first: **/bin**. Some OS/390 shell commands run other shell commands and utilities by name; they expect to get the OS/390 UNIX version of that command and may not work correctly if a program that has the same name is found first in another directory.

Adding Your Working Directory to the Search Path

You can have the shell search your working directory for commands (in addition to the standard directories that contain commands). As an example, suppose you have different directories containing the source code for different programs. In each directory, you create a shell script named **compile** that compiles all the source modules of the program in that directory. To compile a particular program, enter **cd** to change to the appropriate directory and then enter:

```
compile
```

The shell searches the working directory, finds the **compile** shell script, and runs it.

You can add your working directory to your search path by one of these methods:

- Putting in an entry without a name
- Using a period (**.**) for the working directory.

For example, both of these specify that the working directory should be searched after **/bin** but before **/usr/local**:

```
PATH='/bin::/usr/local'    #no name
PATH='/bin:.:/usr/local'   #using a period
```

Both of these say that your working directory should be searched before anything else:

```
PATH=':/bin:/usr/local'    #no name
PATH='.:/bin:/usr/local'   #using a period
```

Both of these say that your working directory should be searched after everything else:

```
PATH='/bin:/usr/local:'    #no name, ends in a colon
PATH='/bin:/usr/local:.'   #using a period
```

The best way to specify search paths is to put them into your **.profile** file. That way, they are set up every time you log into the shell.

Checking the Search Path Used for a Command

With aliases and search paths, it can be easy to lose track of what is actually executed when you enter a command. The **type** command can tell you which file is executed if you enter a command line that begins with a specific command. For example:

```
type date
```

tells you:

```
date is /bin/date
```

and the command:

```
type jobs
```

tells you:

```
jobs is a built-in command
```

You can figure out how the search path works and what effect aliases have.

Customizing the FPATH Search path: the FPATH Variable

The **FPATH** variable contains a list of directories that the OS/390 shell searches to find executable functions. Directories in this list are separated by colons. **sh** searches each directory in the order specified in the list until it finds a matching function. **FPATH** should specify only directories where the only executable files are function definitions.

Customizing the DLL Search Path: The LIBPATH Variable

If you use a utility that uses a dynamic link library (DLL) —for example, **dbx**— you can set up the search path for the DLL with the LIBPATH variable. If this variable is not set, your working directory is searched for the DLL. The default setting shipped in **/samples/profile** is:

```
LIBPATH=/lib:/usr/lib:.
```

Improving the Performance of Shell Scripts

To improve the performance of shell scripts, set the **_BPX_SPAWN_SCRIPT** environment variable to a value of YES.

If **_BPX_SPAWN_SCRIPT=YES** is not already placed in **/etc/profile**, you can put it in your **$HOME/.profile**.

Here is what the variable does: if the spawn callable service determines that a file is not an HFS executable or a REXX exec, this setting causes spawn to run the file as a shell script directly. In the default processing, however, if the spawn callable service determines that a file is not an HFS executable or a REXX exec, the spawn fails with ENOEXEC and the shell then forks another process to run the input shell script. Setting this variable to YES eliminates the additional overhead of the fork.

You may want to set the variable to NO when you are running a non-shell application. For example, if an application does not support shell script invocations, set the variable to NO. Likewise, if an application is in test mode and the ENOEXEC returned would be a useful indication of an error in the format of the target executable file, set the variable to NO.

Changing the Locale in the Shell

The default locale for the shell and utilities is C. If you want to change the locale, read these topics:

- "Advantages of a Locale Compatible with the MVS Code Page"
- "Advantages of a Locale Generated with Code Page IBM-1047" on page 46
- "Changing the Locale Setting in Your Profile" on page 46
- "The LC_SYNTAX Environment Variable" on page 47
- "The LOCPATH Environment Variable" on page 49

For additional information on locale and **LC_SYNTAX**, see *OS/390 Language Environment Programming Guide*.

Advantages of a Locale Compatible with the MVS Code Page

Running the shell and utilities in a locale whose code page matches the code page you are using in MVS (which may not be compatible with code page IBM-1047 with respect to the EBCDIC variant characters) has several advantages:

- Converting data from a given country's native code page to IBM-1047 is no longer required. This may enhance interoperability with other non-OS/390 UNIX components of MVS.
- Remapping your keyboard is unnecessary.

Customizing for a Locale Not Based on Code Page IBM-1047

If you select a locale that is not based on code page IBM-1047 and you use the utilities **lex**, **mailx**, **make**, and **yacc**, there is a further customizing step. These utilities expect all their input files, both system files and user-created files, to be in the same code page. So, for example, if you select the German locale De_DE.IBM-273, these utilities expect the files they process to be in code page IBM-273. Because system files are in code page IBM-1047, you need to use **iconv** to convert the following system files to the code page used by your selected locale:

Utility	File
lex	**/etc/yylex.c**
mailx	**/etc/mailx.rc**
make	**/etc/startup.mk**
yacc	**/etc/yyparse.c**

Advantages of a Locale Generated with Code Page IBM-1047

On the other hand, you may prefer using one of the locales compatible with IBM-1047, but not compatible with the MVS code page if:

- You already use one of the IBM-1047 locales and have made an investment in data conversion and keyboard remapping.
- You have a requirement to run, in your shell environment, strictly standards-compliant applications or other applications that do *not* use **LC_SYNTAX**. If you want to use a single compiled and link-edited instance of a program in multiple locales, such a program is guaranteed to work in multiple locales only if IBM-1047 locales are used.
- You have shell scripts that are used in multiple locales. Having different users operating in various locales that are not generated from code page IBM-1047 requires multiple copies of a shell script, one for each different locale's code page.

There are other important code page conversion considerations when the shell uses code page 1047 and MVS does not; see "Appendix E. Code Page Conversion when the Shell and MVS Have Different Locales" on page 345 for that information.

Changing the Locale Setting in Your Profile

To change the locale, you set the value for the **LC_ALL** variable and export it. This variable overrides any values for locale specified for the **LC_** variables such as **LC_COLLATE**, **LC_MESSAGES**, and **LC_SYNTAX**, but it does not override **LC_CTYPE**.

If you change **LC_ALL** to a new locale, and OS/390 UNIX messages are provided in that language, change the **LANG** variable setting to match the **LC_ALL** setting. Currently, OS/390 UNIX messages are shipped in English, Kanji, and Simplified Chinese. If you do not change **LANG**, the messages will be in English.

If OS/390 UNIX messages are not provided in your language then changing **LANG** by itself will have no effect. However, although messages are not supplied in your language, the OS/390 UNIX messages that are displayed in English will use your national language characters and should display correctly on your terminals.

When you change the locale, the shell and utilities run in the new locale, but the shell locale category **LC_CTYPE** stays in the POSIX locale. This can affect parsing and shell expansion, and cause unpredictable behavior. In order to avoid this problem, after you change locale you must overwrite the current shell by issuing the **exec sh -L** command. The new shell will correctly interpret the proper character set for the new locale.

If you place an `export LC_ALL=`*`localename`* statement in your login profile, or if one has been placed in **/etc/profile**, make sure it is followed with **exec sh -L** and protect that with **tty -s** as shown in the example below. If you don't protect it with the **tty -s** test, then BPXBATCH SH *command* will not run the command.

If you use **exec sh -L**, there are two situations that you must take into account:

1. Loop control; you only want the **exec sh -L** executed the first time.
2. If you plan to use BPXBATCH or OSHELL (which calls BPXBATCH) with national language support, you need to define the LANG and LC_ALL variables in a file for BPXBATCH to use. See "Defining an Environment Variable File for BPXBATCH" on page 162 for more information.

If your **/etc/profile** has been set up for the proper locale, you only need to change your .profile if you want a different locale than already set up as the default. For more information on setting up locale and messages, see "Customizing for Your National Code Page" in the *OS/390 UNIX System Services Planning*.

Examples: Changing Locale

For example, say you are using OMVS, the 3270 terminal interface. If your /etc/profile is not set up for your locale and LANG, then in order to work in a locale such as Danish, you should add this to your **.profile** file:

```
if test -z "$LOCALE_SWITCH" && tty -s
then
  echo " - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - "
  echo " - Logon shell will now be invoked to reflect        - "
  echo " - code page IBM-277                                 - "
  echo " - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - "
  LOCALE_SWITCH=EXECUTED
  LANG=C
  LC_ALL=Da_DK.IBM-277
  export LANG LC_ALL LOCALE_SWITCH
  #Issue chcp if not using OMVS command
   if test "$_BPX_TERMPATH" ! "OMVS"
   then
   chcp -a ISO8859-1 -e IBM-277
   fi
exec sh -L
else
  echo " - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - "
  echo " -  Welcome to OS/390 UNIX System Services           - "
  echo " - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - "
fi
```

If you want your messages displayed in a different language than that specified in the system-wide **/etc/profile**, you have to modify your **.profile** accordingly. For more information, see "Customizing the Language of Your Messages" on page 49.

For a list of the OS/390 UNIX locales (and their locale object names) and locale source files, see "Appendix G. Locale Objects, Source Files, and Charmaps" on page 353.

The LC_SYNTAX Environment Variable

There are 13 "variant" characters in the POSIX portable character set whose encoding may vary on various EBCDIC code pages:

- Right brace (})
- Left brace ({)
- Backslash (\)
- Right square bracket (])
- Left square bracket ([)
- Circumflex (^)
- Tilde (~)
- Exclamation point (!)
- Pound sign (#)
- Vertical bar (|)
- Dollar sign ($)
- Commercial at-sign (@)
- Accent grave (`)

Prior to MVS SP Release 5.2.2, the OS/390 shell and utilities required that all data in the hierarchical file system (HFS) be encoded in one of three code pages:

IBM-1047, IBM-1027, or IBM-939. Any data moved into the HFS from a workstation or from an MVS dataset often had to be converted to one of code pages IBM-1047, IBM-1027, or IBM-939 before it could be processed by the shell. Similarly, to ensure that any variant characters keyed in at the terminal had the correct encoding, you had to either use the conversion option of the OMVS command or customize your keyboard.

Now, however, the shell can process data in additional EBCDIC code pages, not just the three code pages previously supported. When you specify a locale with the **LC_ALL** variable, the **LC_SYNTAX** environment variable is set. The shell uses the **LC_SYNTAX** environment variable to determine the code points to use for the 13 variant characters. This means the shell can dynamically adapt to the code page of the current locale.

Applications that use **LC_SYNTAX** will work in multiple locales using multiple code pages. To be sensitive to the 13 variant characters, an application must be enabled to use **LC_SYNTAX**. For information on how to do this, see *OS/390 C/C++ Programming Guide*.

LC_SYNTAX—An Example

For example, consider the **echo** command and its use of the backslash (\) character. The backslash is one of the 13 variant characters. The following command:

```
echo 'this is\nreal handy'
```

produces the following output at the terminal:

```
this is
real handy
```

echo finds and converts the `\n` in the input to a <newline> character in the output. To do this, **echo** must know the encoding for the backslash character in the current user's environment—in this case, the character generated by the user's terminal when the backslash key is pressed.

A 3270 terminal operating in the USA locale En_US.IBM-037 (code page IBM-037) generates X'E0' for the backslash, while a 3270 terminal operating in the German locale De_DE.IBM-273 (code page IBM-273) generates X'EC'. The **LC_SYNTAX** locale category provides this locale-specific hexadecimal encoding information to **echo** and the other utilities.

When the USA user runs in locale En_US.IBM-037, **echo** determines from the LC_SYNTAX information in this locale that the expected encoding for backslash is X'E0'. Likewise, when the German user runs in locale De_DE.IBM-273, **echo** determines from the LC_SYNTAX information in this locale that the expected encoding for backslash is X'EC'.

Limitations

The **LC_SYNTAX** setting does not affect:

- REXX execs.
- The ISPF shell (ISHELL). ISHELL runs in the locale that MVS is using, and therefore this could be different from the shell locale.
- Shell scripts: The code page in which a shell script is encoded must match the code page of the locale in which it is run. For a shell script to be shared by

multiple users, they must all be in a locale that uses the same code page as the code page in which the shell script is encoded.

If you have different users operating in various locales, you need multiple copies of a shell script, one for each different locale code page. You can use the **iconv** command to convert a shell script from one code page to another.

The LOCPATH Environment Variable

LOCPATH is an environment variable that tells the **setlocale()** function the name of the directory from which to load locale object files. If LOCPATH is not defined, the default directory **/usr/lib/nls/locale** is searched. LOCPATH is similar to the PATH environment variable; it contains a list of HFS directories separated by colons. For detailed information on how **setlocale()** searches for locale object files, see the description of **setlocale()** in *OS/390 C/C++ Run-Time Library Reference*.

Customizing the Language of Your Messages

If you want your messages displayed in a different language than that specified in the system-wide **/etc/profile**, add this line to your **.profile**:

```
export LANG=your_language
```

your_language is the first part of the locale name listed in Table 18 on page 353—for example, Ja_JP in the locale name JA_JP.IBM-939. Currently, OS/390 UNIX ships messages in English, Kanji and Simplified Chinese.

Setting Your Local Time Zone

The shell and utilities assume that the times stored in the file system and returned by the operating system are stored using the Greenwich Mean Time (GMT) or Universal Time Coordinated (UTC) as a universal reference. In the system-wide **/etc/profile**, the **TZ** environment variable maps that reference time to the local time specified with the variable. You can use a different time zone by setting the **TZ** variable in your ·**profile**.

The three primary fields in the time zone specification are:

1. The local standard time, abbreviated—for example, EST or MSEZ.
2. The time offset west from the universal reference time, typically specified in hours (minutes and seconds are optional). A minus sign (–) indicates an offset east of the universal reference time.
3. The daylight savings time zone, abbreviated—for example, EDT. If this and the first field are identical or this value is missing, daylight savings time conversion is disabled. Optionally, you can specify an additional rule that indicates when Daylight Savings Time starts and ends.

For example, if you want to set your time zone to Eastern Standard Time (EST)and export it, you would specify:

```
export TZ="EST5EDT"
```

- EST is Eastern Standard Time, the local time zone.
- The standard time zone is 5 hours west of the universal reference time.
- EDT is Eastern Daylight Savings time zone.

For complete information on how to specify the local time zone, see *OS/390 UNIX System Services Command Reference*.

Building a STEPLIB Environment: The STEPLIB Environment Variable

Traditionally, some MVS users have preferred to alter the search order for MVS executable files when they are running a new or test version of an application program, such as a runtime library. To do this, they code a STEPLIB DD statement on the JCL used to run the application. Accessed ahead of LINKLIB or LPALIB, a STEPLIB is a set of private libraries where the new or test version of the application is stored.

The STEPLIB environment variable provides the ability to use a STEPLIB when running an HFS executable file. This variable is used to determine how to set up the STEPLIB environment for an executable file. The STEPLIB environment variable should always be exported.

You can set the variable in one of three ways:

STEPLIB=CURRENT
: Passes on any currently active TASKLIB, STEPLIB, or JOBLIB allocations from the invoker's MVS program search order environment to the environment created for the executable file to run in. Any STEPLIB environment in the invoker's process image is re-created in the new process image for the executable file when the file is invoked. This is the default value that is set if no STEPLIB variable is specified.

  If an application uses **fork()**, **spawn()**, or **exec()**, then the STEPLIB data sets must be cataloged.

STEPLIB=NONE
: Specifies that no STEPLIB environment should be set up for executable files.

STEPLIB=DSN1:DSN2:DSN3
: Sets up a library search order for the STEPLIB, in the order that the data sets are specified. You can specify up to 255 fully qualified data set names, separated by colons—for example:

  ```
  export STEPLIB=SMITH.C.LOADLIB:SMITH.PL1.LOADLIB
  ```

  The specified data sets must be cataloged MVS load libraries that you have security access to. The data sets specified here are built into a STEPLIB environment for the executable file.

Restrictions on STEPLIB Data Sets

For executable files that have the set-user-ID or set-group-ID bit set, there are restrictions on the data sets that can be built into the STEPLIB environment for the file to run in. The system programmer maintains a STEPLIB sanction list of data sets that can be included in the STEPLIB environment for such executable files. Only data sets on that list are built into the STEPLIB environment for such files. If you need a data set added to the list, contact your system programmer. For more information on the STEPLIB sanction list, see *OS/390 UNIX System Services Planning*.

Setting Options for a Shell Session

The **set** command lets you set options, or flags, for your shell session. These flags control the way the shell handles certain situations. To display the shell flags that are currently set, type `set –o`. To turn an option on, enter:

`set –o` *name*

where *name* is the name of the option you want to turn on. If you want an option turned on for every shell session, put the **set** command in your login script (the script specified on the **ENV** variable).

To turn an option off, enter:

```
set +o name
```

Contrary to what you might expect, - means *on,* and + means *off*.

The following discussion highlights some of the options you may find useful. For all the options, see the description of **set** in *OS/390 UNIX System Services Command Reference* .

Exporting Variables

The command:

```
set -o allexport
```

indicates that you want to *export*—that is, pass to a child process or subsequent command—every variable that is assigned a value. This command exports all variables that currently have values, plus all variables assigned a value in the future.

Controlling Redirection

The command:

```
set -o noclobber
```

indicates that you do not want the **>** redirection operator to overwrite existing files. When this option is on and you specify the construct **>***file*, the redirection works only if *file* does not already exist. If you have this option on and you really do want to redirect output into an existing file, you must use **>|***file* (with an "or" bar after the **>**) to indicate output redirection. See Using a Wildcard Character to Specify Filenames for more information.

Preventing Wildcard Character Expansion

The command:

```
set -o noglob
```

tells the shell not to expand wildcard characters in filenames. This command is occasionally useful if you are entering command lines that contain a number of characters that would normally be expanded. See "Using a Wildcard Character to Specify Filenames" on page 80 for a discussion of wildcard characters.

Displaying Input from a File

The command:

```
set -o verbose
```

tells the shell to display its input on the screen as the input is read. This command lets you keep track of material that comes from a file.

Running a Command in the Current Environment

The command:

```
set -o pipecurrent
```

causes the shell to run the last command of a pipeline in the current environment.

Displaying Current Option Settings

The command:

```
set -o
```

displays all current option settings. The display of each option is preceded by one of these:

- `-o` to indicate the option is enabled
- `+o` to indicate the option is disabled

Chapter 5. Customizing the tcsh Shell

If you are interested in using the tcsh shell, read this chapter and:

- "Chapter 7. Working with tcsh Shell Commands" on page 95
- "Chapter 9. Writing tcsh Shell Scripts" on page 139

You can personalize your use of the tcsh shell. This chapter covers these topics:

- Understanding and modifying your startup files
- Understanding shell variables
- Customizing the search path for commands with the **PATH** variable
- Improving the performance of shell scripts
- Changing the locale
- Customizing the language of messages
- Setting the time zone
- Building a STEPLIB environment
- Setting options for a shell session

Understanding the Startup Files

When you start the tcsh shell, it uses information in several files to determine your particular needs or preferences as a user. The files are accessed in the following order:

1. **/etc/csh.cshrc**
2. **/etc/csh.login**
3. **$HOME/.tcshrc**
4. **$HOME/.cshrc**
5. **$HOME/.history**
6. **$HOME/.login**
7. **$HOME/.cshdirs**

Settings established in a file accessed earlier can be overwritten by the settings in a file accessed later.

The **/etc/csh.cshrc** file contains systemwide settings that are common to all shell users. It is used for setting shell variables and defining command aliases. Usually, it will set environment variables such as PATH.

The **/etc/csh.login** file is a systemwide file that is only executed by tcsh login shells, and is used for setting environment variables such as TERM. Opening messages are typically placed here.

The **/$HOME/.tcshrc** file contains settings that may be customized for an individual shell user. It is used for setting shell variables and defining command aliases. Here, users can set variables that are different than the system defaults set in the systemwide profiles.

The **/$HOME/.cshrc** file is included for compatibility with C-Shell users, and is read only if **/$HOME/.tcshrc** does not exist. It contains the same type of settings as **/$HOME/.tcshrc**.

The /$HOME/.history file is read by login shells to initialize the history list. It is created by the shell, based on the setting of certain shell variables.

The **/$HOME/.login** file is only executed by tcsh login shells, and is used for setting environment variables that have been customized for an individual user. It usually contains commands that affect a user's terminal settings.

Typically, your **.login** file might contain the following:

```
# set TERM environment variable
setenv TERM vt220

# set DISPLAY environment variable
setenv DISPLAY mymachine.mydomain.com:0
```

Figure 12. A Sample .login

The **$HOME/.cshdirs** file is read by login shells to initialize the directory stack. It is created by the shell, based on the setting of certain shell variables.

The systemwide startup files (located in /etc) are modified by system administrators to contain settings that should pertain to all users. The startup files in a user's home directory ($HOME/. . .) can be altered to suit specific user preferences, with the exception of **$HOME/.history** and **$HOME/.cshdirs**, which are created by the shell. A user can ″unset″ or ″unalias″ anything that was defined in a systemwide startup file.

Quoting Variable Values

When you have blanks in a variable value, you need to enclose it in quotes. The quotes tell the shell to treat blanks as literals and not delimiters. Single quotes are more "serious" about this than are double quotes:

- Single quotes preserve the meaning of (that is, treat literally) all characters.
- Double quotes still allow certain characters ($, ` (backquote), and \ (backslash)) to be expanded. This is important if you want variable expansion. For example, see how the $ is handled here:

  ```
  setenv HOMEMSG "Using $HOME as Home Directory"
  ```

  If your home directory were set to **/u/user**, the following:

  ```
  echo $HOMEMSG
  ```

  would display:

  ```
  Using /u/user as home directory
  ```

  If, instead, you enclosed the variable value in single quotes, like this:

  ```
  setenv HOMEMSG 'Using $HOME as home directory'
  ```

  the following:

  ```
  echo $HOMEMSG
  ```

  would display:

  ```
  Using $HOME as home directory
  ```

  As you can see, the $ is not expanded.

Changing Variable Values Dynamically

You can also change any of these values for the duration of your session (or until you change them again). You enter the name of the environment or shell variable and equate it to a new value. For example:

```
set prompt='+>'
```

changes the command prompt string to +>.

Understanding Shell Variables

You can display the shell's variables and their values by entering this command:

```
set
```

or

```
set -r
```

set —r displays readonly shell variables.

You may see many variables that you don't recognize. These are *built-in*, or *predefined*, variables that are set up with default values when you start the shell.

You can customize the built-in variables by setting their value in your **.tcshrc** file.

Only the shell variables that are defined in the **.tcshrc** file are available to shell scripts and commands invoked from the shell. Environment variables are inherited by subshells, and can be displayed by entering either of these commands:

```
setenv
printenv
```

You can display the value of a single variable with the **echo** command or the **printenv** command. For example, either of these commands

```
echo $HOME
```

```
printenv $HOME
```

displays the current value of the **HOME** variable.

In general, **echo** displays the current values of all its arguments, after any shell processing has taken place. For example, consider:

```
echo *.doc
```

The shell first expands the wildcard character *****. This produces the names of every file in the working directory that has the suffix **.doc**. So the output of **echo** is a list of all such files. And if there are no filenames ending in **.doc**, the command output is just `*.doc`.

For more information about shell variables,

- Built-in variables are listed in a table in the **tcsh** command description in *OS/390 UNIX System Services Command Reference*.
- There is an appendix that lists shell variables in *OS/390 UNIX System Services Command Reference*.

Customizing Your Shell Environment: The .tcshrc File

So far, we have discussed customization that is set up inside your **.login** file. However, the shell reads this file only when you log into the shell or when you enter the **tcsh** command with the **–l** option. Note that the option is a lowercase ″L″.

To always have a customized shell session, you need to have a special shell script that customizes your shell variables each time you start the shell; this is the purpose of the **.tcshrc** file (also known as a startup script).

For example, you might put all your alias definitions and other setup instructions into this file. You want these instructions run when your shell starts after you login and whenever you explicitly create the shell during a session (for example, as a child shell to run a shell script).

Below is a sample **.tcshrc** file:

```
# ================================================================
#                       path shell variable
#                       -------------------
# Lists directories in which to look for executable commands.
# ================================================================
#set path = ( /bin /usr/local/bin /usr/bin )

# test if we are an interactive shell
if ($?prompt) then
# ================================================================
#                       prompt shell variable
#                       ---------------------
# The string which is printed before reading each command from the
# terminal.  Currently set to display hostname, and current working
# directory.
# ================================================================
set prompt = "%m:%~> "

# ================================================================
#                       rmstar shell variable
#                       ---------------------
# If set, the user is prompted before 'rm *' is executed.
# ================================================================
set rmstar

# ================================================================
#                       noclobber shell variable
#                       ------------------------
# If set, output redirection will not overwrite existing files.
# ================================================================
#set noclobber

# ================================================================
# source complete.tcsh
# ================================================================
if (′filetest -e /etc/complete.tcsh′) then
        source /etc/complete.tcsh
endif
endif  # interactive shell

# ================================================================
# set up useful aliases
# ================================================================
alias m more
```

Figure 13. A Sample .tcshrc

Customizing the Search Path for Commands: The PATH Variable

Command interpreters usually have to *search* for a file that contains the command you want to run. When using the shell, you tell the shell where to search for a command. Essentially, the shell uses a list of directories in which commands may be found. This list is specified in your **PATH** variable in your **etc/csh.cshrc** file. The list could be called your *search path,* because it tells the shell where you want to search.

You can set up a search path with a command of the form:

```
setenv path 'dir:dir:...'
```

or,

```
set path=(dir1 dir2)
```

For example, you might enter:

```
setenv path '/bin:/usr/bin:/usr/macneil/bin:/usr/games:/usr'
```

The shell then searches the directories in the following order, when looking for commands or shell scripts:

1. **/bin**
2. **/usr/bin**
3. **/usr/macneil/bin**
4. **/usr/games**
5. **/usr**

As soon as the shell finds a file with an appropriate name, it runs that file.

Because the shell runs a command as soon as it finds a file with an appropriate name, pay close attention to the order in which you list directory names in your search path. For example, the previous search path specifies the **/bin** directory (where OS/390 shell commands are stored) before the **/usr/bin** directory.

If you set up your **PATH** incorrectly, you could get the wrong command. You should generally search the shell commands directory first: **/bin**.

Adding Your Working Directory to the Search Path

You can have the shell search your working directory for commands (in addition to the standard directories that contain commands). As an example, suppose you have different directories containing the source code for different programs. In each directory, you create a shell script named **compile** that compiles all the source modules of the program in that directory. To compile a particular program, enter **cd** to change to the appropriate directory and then enter:

```
compile
```

The shell searches the working directory, finds the **compile** shell script, and runs it.

You can add your working directory to your search path by one of these methods:

- Putting in an entry without a name
- Using a period (**.**) for the working directory.

For example, both of these specify that the working directory should be searched after **/bin** but before **/usr/local**:

```
setenv path '/bin::/usr/local'    #no name
setenv path '/bin:.:/usr/local'   #using a period
```

Both of these say that your working directory should be searched before anything else:

```
setenv path ':/bin:/usr/local'    #no name
setenv path '.:/bin:/usr/local'   #using a period
```

Both of these say that your working directory should be searched after everything else:

```
setenv path '/bin:/usr/local:'    #no name, ends in a colon
setenv path '/bin:/usr/local:.'   #using a period
```

The best way to specify search paths is to put them into your **.tcshrc** file. That way, they are set up every time you log into the shell.

Checking the Search Path Used for a Command

With aliases and search paths, it can be easy to lose track of what is actually executed when you enter a command. The **which** command can tell you which file is executed if you enter a command line that begins with a specific command. The **where** command can tell you where versions of the command are located. For example:

```
which kill
```

tells you:

```
kill: shell built-in command.
```

and the command:

```
where kill
```

tells you:

```
kill is a shell built-in
/bin/kill
```

Customizing the DLL Search Path: The LIBPATH Variable

If you use a utility that uses a dynamic link library (DLL) —for example, **dbx**— you can set up the search path for the DLL with the LIBPATH variable. If this variable is not set, your working directory is searched for the DLL. The default setting shipped in **/samples/login** is:

```
setenv LIBPATH "/lib:/usr/lib:."
```

Changing the Locale in the Shell

The default locale for the shell and utilities is C. If you want to change the locale, read the topics below.

For additional information on locale and **LC_SYNTAX**, see *OS/390 Language Environment Programming Guide*.

Advantages of a Locale Compatible with the MVS Code Page

Running the shell and utilities in a locale whose code page matches the code page you are using in MVS (which may not be compatible with code page IBM-1047 with respect to the EBCDIC variant characters) has several advantages:

- Converting data from a given country's native code page to IBM-1047 is no longer required. This may enhance interoperability with other non-OS/390 UNIX components of MVS.
- Remapping your keyboard is unnecessary.

Customizing for a Locale Not Based on Code Page IBM-1047

If you select a locale that is not based on code page IBM-1047 and you use the utilities **lex**, **mailx**, **make**, and **yacc**, there is a further customizing step. These utilities expect all their input files, both system files and user-created files, to be in the same code page. So, for example, if you select the German locale De_DE.IBM-273, these utilities expect the files they process to be in code page IBM-273. Because system files are in code page IBM-1047, you need to use **iconv** to convert the following system files to the code page used by your selected locale:

Utility	File
lex	/etc/yylex.c

mailx /etc/mailx.rc

make /etc/startup.mk

yacc /etc/yyparse.c

Advantages of a Locale Generated with Code Page IBM-1047

On the other hand, you may prefer using one of the locales compatible with IBM-1047, but not compatible with the MVS code page if:

- You already use one of the IBM-1047 locales and have made an investment in data conversion and keyboard remapping.
- You have a requirement to run, in your shell environment, strictly standards-compliant applications or other applications that do *not* use **LC_SYNTAX**. If you want to use a single compiled and link-edited instance of a program in multiple locales, such a program is guaranteed to work in multiple locales only if IBM-1047 locales are used.
- You have shell scripts that are used in multiple locales. Having different users operating in various locales that are not generated from code page IBM-1047 requires multiple copies of a shell script, one for each different locale's code page.

There are other important code page conversion considerations when the shell uses code page 1047 and MVS does not; see "Appendix E. Code Page Conversion when the Shell and MVS Have Different Locales" on page 345 for that information.

Changing the Locale Setting in Your Profile

To change the locale, you set the value for the **LC_ALL** variable. This variable overrides any values for locale specified for the **LC_** variables such as **LC_COLLATE**, **LC_MESSAGES**, and **LC_SYNTAX**, but it does not override **LC_CTYPE**.

If you change **LC_ALL** to a new locale, and OS/390 UNIX messages are provided in that language, change the **LANG** variable setting to match the **LC_ALL** setting. Currently, OS/390 UNIX messages are shipped in English, Kanji, and Simplified Chinese. If you do not change **LANG**, the messages will be in English.

If OS/390 UNIX messages are not provided in your language then changing **LANG** by itself will have no effect. However, although messages are not supplied in your language, the OS/390 UNIX messages that are displayed in English will use your national language characters and should display correctly on your terminals.

When you change the locale, the shell and utilities run in the new locale, but the shell locale category **LC_CTYPE** stays in the POSIX locale. This can affect parsing and shell expansion, and cause unpredictable behavior. In order to avoid this problem, after you change locale you must overwrite the current shell by issuing the **exec tcsh -l** command. The new shell will correctly interpret the proper character set for the new locale.

If you place an `setenv LC_ALL` *`localename`* statement in your login profile, or if one has been placed in **/etc/csh.login**, make sure it is followed with **exec tcsh -l** and protect that with **tty -s** as shown in the example below. If you don't protect it with the **tty -s** test, then BPXBATCH SH *command* will not run the command.

If you use **exec tcsh -l**, there are two situations that you must take into account:

1. Loop control; you only want the **exec tcsh -l** executed the first time.

2. If you plan to use BPXBATCH or OSHELL (which calls BPXBATCH) with national language support, you need to define the LANG and LC_ALL variables in a file for BPXBATCH to use. See “Defining an Environment Variable File for BPXBATCH” on page 162 for more information.

If your **/etc/csh.login** has been set up for the proper locale, you only need to change your .login if you want a different locale than already set up as the default. For more information on setting up locale and messages, see “Customizing for Your National Code Page” in the *OS/390 UNIX System Services Planning*.

Examples: Changing Locale

For example, say you are using OMVS, the 3270 terminal interface. If your /etc/csh.login is not set up for your locale and LANG, then in order to work in a locale such as Danish, you should add this to your **.login** file:

```
tty -s
set tty_rc=$status
if (($?LOCALE_SWITCH == 0 ) && ($tty_rc == 0)) then
        echo "-------------------------------------------------"
        echo "- Logon shell will now be invoked to reflect   -"
        echo "- code page IBM-277                            -"
        echo "-------------------------------------------------"
        setenv LOCALE_SWITCH EXECUTED
        setenv LANG C
        setenv LC_ALL Da_DK.IBM-277
        # Issue chcp if not using OMVS command
        if ($?_BPX_TERMPATH != "OMVS"  ) then
          chcp -a ISO8859-1 -e IBM-277
        endif
        exec tcsh -l
endif
unset tty_rc
```

If you want your messages displayed in a different language than that specified in the system-wide **/etc/csh.login**, you have to modify your **.login** accordingly.

For a list of the OS/390 UNIX locales (and their locale object names) and locale source files, see “Appendix G. Locale Objects, Source Files, and Charmaps” on page 353.

The LC_SYNTAX Environment Variable

There are 13 “variant” characters in the POSIX portable character set whose encoding may vary on various EBCDIC code pages:

- Right brace (})
- Left brace ({)
- Backslash (\)
- Right square bracket (])
- Left square bracket ([)
- Circumflex (^)
- Tilde (~)
- Exclamation point (!)
- Pound sign (#)
- Vertical bar (|)
- Dollar sign ($)
- Commercial at-sign (@)
- Accent grave (`)

When you specify a locale with the **LC_ALL** variable, the **LC_SYNTAX** environment variable is set. The shell uses the **LC_SYNTAX** environment variable to determine

the code points to use for the 13 variant characters. This means the shell can dynamically adapt to the code page of the current locale.

Applications that use **LC_SYNTAX** will work in multiple locales using multiple code pages. To be sensitive to the 13 variant characters, an application must be enabled to use **LC_SYNTAX**. For information on how to do this, see *OS/390 C/C++ Programming Guide*.

LC_SYNTAX—An Example

For example, consider the **echo** command and its use of the backslash (\) character. The backslash is one of the 13 variant characters. When the echo style is **all** or **sysv**, the following command:

```
echo 'this is\nreal handy'
```

produces the following output at the terminal:

```
this is
real handy
```

echo finds and converts the `\n` in the input to a <newline> character in the output. To do this, **echo** must know the encoding for the backslash character in the current user's environment—in this case, the character generated by the user's terminal when the backslash key is pressed.

A 3270 terminal operating in the USA locale En_US.IBM-037 (code page IBM-037) generates X'E0' for the backslash, while a 3270 terminal operating in the German locale De_DE.IBM-273 (code page IBM-273) generates X'EC'. The **LC_SYNTAX** locale category provides this locale-specific hexadecimal encoding information to **echo** and the other utilities.

When the USA user runs in locale En_US.IBM-037, **echo** determines from the LC_SYNTAX information in this locale that the expected encoding for backslash is X'E0'. Likewise, when the German user runs in locale De_DE.IBM-273, **echo** determines from the LC_SYNTAX information in this locale that the expected encoding for backslash is X'EC'.

Limitations

The **LC_SYNTAX** setting does not affect:

- REXX execs.
- The ISPF shell (ISHELL). ISHELL runs in the locale that MVS is using, and therefore this could be different from the shell locale.
- Shell scripts: The code page in which a shell script is encoded must match the code page of the locale in which it is run. For a shell script to be shared by multiple users, they must all be in a locale that uses the same code page as the code page in which the shell script is encoded.

  If you have different users operating in various locales, you need multiple copies of a shell script, one for each different locale code page. You can use the **iconv** command to convert a shell script from one code page to another.

The LOCPATH Environment Variable

LOCPATH is an environment variable that tells the **setlocale()** function the name of the directory from which to load locale object files. If LOCPATH is not defined, the default directory **/usr/lib/nls/locale** is searched. LOCPATH is similar to the PATH environment variable; it contains a list of HFS directories separated by colons. For

detailed information on how **setlocale()** searches for locale object files, see the description of **setlocale()** in *OS/390 C/C++ Run-Time Library Reference*.

Customizing the Language of Your Messages

If you want your messages displayed in a different language than that specified in the system-wide **/etc/.login**, add this line to your **.login**:

```
setenv LANG your_language
```

your_language is the first part of the locale name listed in Table 18 on page 353—for example, Ja_JP in the locale name JA_JP.IBM-939. Currently, OS/390 UNIX ships messages in English, Kanji and Simplified Chinese.

Setting Your Local Time Zone

The shell and utilities assume that the times stored in the file system and returned by the operating system are stored using the Greenwich Mean Time (GMT) or Universal Time Coordinated (UTC) as a universal reference. In the system-wide **/etc/csh.login**, the **TZ** environment variable maps that reference time to the local time specified with the variable. You can use a different time zone by setting the **TZ** variable in your **.login**.

The three primary fields in the time zone specification are:

1. The local standard time, abbreviated—for example, EST or MSEZ.
2. The time offset west from the universal reference time, typically specified in hours (minutes and seconds are optional). A minus sign (–) indicates an offset east of the universal reference time.
3. The daylight savings time zone, abbreviated—for example, EDT. If this and the first field are identical or this value is missing, daylight savings time conversion is disabled. Optionally, you can specify an additional rule that indicates when Daylight Savings Time starts and ends.

For example, if you want to set your time zone to Eastern Standard Time (EST)and export it, you would specify:

```
setenv TZ "EST5EDT"
```

- EST is Eastern Standard Time, the local time zone.
- The standard time zone is 5 hours west of the universal reference time.
- EDT is Eastern Daylight Savings time zone.

For complete information on how to specify the local time zone, see *OS/390 UNIX System Services Command Reference*.

Building a STEPLIB Environment: The STEPLIB Environment Variable

Traditionally, some MVS users have preferred to alter the search order for MVS executable files when they are running a new or test version of an application program, such as a runtime library. To do this, they code a STEPLIB DD statement on the JCL used to run the application. Accessed ahead of LINKLIB or LPALIB, a STEPLIB is a set of private libraries where the new or test version of the application is stored.

The STEPLIB environment variable provides the ability to use a STEPLIB when running an HFS executable file. This variable is used to determine how to set up the STEPLIB environment for an executable file.

You can set the variable in one of three ways:

setenv STEPLIB CURRENT
Passes on any currently active TASKLIB, STEPLIB, or JOBLIB allocations from the invoker's MVS program search order environment to the environment created for the executable file to run in. Any STEPLIB environment in the invoker's process image is re-created in the new process image for the executable file when the file is invoked. This is the default value that is set if no STEPLIB variable is specified.

If an application uses **fork()**, **spawn()**, or **exec()**, then the STEPLIB data sets must be cataloged.

setenv STEPLIB NONE
Specifies that no STEPLIB environment should be set up for executable files.

setenv STEPLIB DSN1:DSN2:DSN3
Sets up a library search order for the STEPLIB, in the order that the data sets are specified. You can specify up to 255 fully qualified data set names, separated by colons—for example:

```
setenv STEPLIB SMITH.C.LOADLIB:SMITH.PL1.LOADLIB
```

The specified data sets must be cataloged MVS load libraries that you have security access to. The data sets specified here are built into a STEPLIB environment for the executable file.

Restrictions on STEPLIB Data Sets

For executable files that have the set-user-ID or set-group-ID bit set, there are restrictions on the data sets that can be built into the STEPLIB environment for the file to run in. The system programmer maintains a STEPLIB sanction list of data sets that can be included in the STEPLIB environment for such executable files. Only data sets on that list are built into the STEPLIB environment for such files. If you need a data set added to the list, contact your system programmer. For more information on the STEPLIB sanction list, see *OS/390 UNIX System Services Planning*.

Setting Variables for a Shell Session

The **set** and **unset** commands let you set and unset variables for your shell session. These variables control the way the shell handles certain situations. To display the shell variables that are currently set, type `set`. To turn an option on, enter:

```
set name
```

where *name* is the name of the option you want to turn on. If you want an option turned on for every shell session, put the **set** command in your **.tschrc** file.

To turn an option off, enter:

```
unset name
```

The following discussion highlights some of the options you may find useful. For all the options, see *set in the tcsh shell* under **set** in *OS/390 UNIX System Services Command Reference* .

Displaying Current Option Settings

The command:

```
set
```

displays all current option settings.

Controlling Redirection

The command:

```
set noclobber
```

indicates that you do not want the **>** redirection operator to overwrite existing files. When this option is on and you specify the construct **>***file*, the redirection works only if *file* does not already exist. If you have this option on and you really do want to redirect output into an existing file, you must use **>|***file* (with an "or" bar after the **>**) to indicate output redirection.

Preventing Wildcard Character Expansion

The command:

```
set noglob
```

tells the shell not to expand wildcard characters in filenames. This command is occasionally useful if you are entering command lines that contain a number of characters that would normally be expanded.

Displaying Input from a File

The command:

```
set xtrace
```

tells the shell to display its input on the screen as the input is read. This command lets you keep track of material that comes from a file.

Displaying Deletion Verification

The command:

```
set rmstar
```

prompts you for deletion verification when you enter the **rm** command in conjunction with the * character.

Files Accessed at Termination

When you terminate the tcsh shell, the following files are read at logout in this order:

1. **/etc/csh.logout**
2. **$HOME/.logout**

Chapter 6. Working with OS/390 Shell Commands

The shell is, above all, a *programmer's* interface. As a result, the shell commands are strongly slanted towards the needs of a programmer. The OS/390 shell has many *general* tools that can help any programmer. In addition, there are a number of commands designed especially for the C programmer.

Specifying Shell Command Options

Most of the commands discussed in this chapter accept options. Shell command options are usually specified by a minus sign (–) followed by a single character. For example, the **ls** command simply lists a directory's contents in multiple columns on your screen. However:

```
ls –F
```

distinguishes between various file types when listing the contents of a directory. (See "Listing Directory Contents" on page 209 for an example.)

```
ls –1
```

lists directory names in a single column.

Options consisting of a minus sign followed by a character are called *simple options*. You specify simple options after the name of the command and before any other arguments for the command (that is, arguments that are not options). For example, you would enter:

```
ls –1 dir1
```

to list the contents of **dir1** in a single column.

Command options and arguments must be typed as singlebyte characters. Additionally, delimiters such as a slash, curly brackets, and parentheses must be typed as singlebyte characters.

The order of options and arguments is important. If you enter:

```
ls dir1 –F
```

ls lists the contents of **dir1** and then tries to list the contents of the directory, or attributes of the file, called **–F**.

As a special notation, most OS/390 shell commands let you specify a double minus sign (--) to separate the options from the nonoption arguments; -- means that there are no more options. Thus, if you really have a directory named **–F**, you could enter:

```
ls -- –F
```

to list the contents of that directory or the file attributes.

The OS/390 shell gives you a shorthand way to specify more than one simple option to a command. For example, **–t** and **–v** are both simple options that you can specify with the **cat** command. (To find out what these options do, read the description of **cat** in *OS/390 UNIX System Services Command Reference*.) You could enter:

```
cat –t –v file
```

or you could combine the two options into:

```
cat –tv file
```

The order of the options is not important:

```
cat –vt file
```

is equivalent to the previous version of the command.

Specifying Options with Accompanying Arguments

In addition to simple options, some commands accept options that have accompanying arguments. Such options look like simple options followed by additional information. The argument may be a number, a string, the name of a file, or something else.

For example, if you read the description of **ps** in *OS/390 UNIX System Services Command Reference*, you will see that **ps** accepts an argument of the form:

```
–u userlist
```

When *OS/390 UNIX System Services Command Reference* shows part of a command line in *italics*, the italicized material is just a placeholder; when you actually use the command, you should fill in something else in its place. In this case, the *userlist* should be a string of one or more UID numbers or login names separated by commas and enclosed in single quotes. In the command:

```
ps –u 'macneil,wellie1'
```

the *userlist* string is *macneil,wellie1*. (If the string does not contain spaces, tabs, or other special characters, you can actually omit the enclosing single quotes, but the command is often easier to read if you use quotes anyway.) When executed, **ps** displays information for the specified users.

Help for Shell Command Usage

If you incorrectly specify a command, a usage note for the command is displayed. The usage note displays the proper format for the command. Often you can display a usage note deliberately if you specify the command with a *-?* option.

For online help information about a command, see “Online Help” on page 89.

Understanding Standard Input, Standard Output, and Standard Error

Once a command begins running, it has access to three files:

1. It reads from its *standard input* file. By default, standard input is the keyboard.
2. It writes to its *standard output* file.
   - If you invoke a shell command from the shell, a C program, or a REXX program invoked from TSO READY, standard output is directed to your terminal screen by default.
   - If you invoke a shell command, REXX program, or C program from the ISPF shell, standard output cannot be directed to your terminal screen. You can specify an HFS file or use the default, a temporary file.
3. It writes error messages to its *standard error* file.
   - If you invoke a shell command from the shell or from a C program or from a REXX program invoked from TSO READY, standard error is directed to your terminal screen by default.

- If you invoke a shell command, REXX program, or C program from the ISPF shell, standard error cannot be directed to your terminal screen. You can specify an HFS file or use the default, a temporary file.

  If the standard output or standard error file contains any data when the command completes, the file is displayed for you to browse.

Using the Shell:

In the shell, the names for these files are:
- **stdin** for the *standard input* file.
- **stdout** for the *standard output* file.
- **stderr** for the *standard error* file.

The shell sometimes refers to these files by their *file descriptors,* or identifiers:
- `0` for **stdin**
- `1` for **stdout**
- `2` for **stderr**

For more information about the file descriptors that the shell supports, see the **sh** command description in *OS/390 UNIX System Services Command Reference*.

Using TSO/E:

When you are invoking the BPXBATCH utility, you can specify these standard files in MVS DD statements, TSO/E ALLOCATE commands, or DYNALLOC macros using the ddnames:
- STDIN for standard input
- STDOUT for standard output
- STDERR for standard error

For more information about BPXBATCH, see “The BPXBATCH Utility” on page 161.

Using ISPF:

When you run shell commands, REXX programs, and C programs from the ISPF shell, **stdout**, and **stderr** cannot be directed to your terminal. You can specify an HFS file, or use the default—a temporary file. If it has any contents, the file is displayed for you to browse when the command or program completes.

Redirecting Command Output to a File

Commands entered at the command line typically use the three standard files described in the previous section, but you can redirect the output for a command to a file you name. If you redirect output to a file that does not already exist, the system creates the file automatically.

Most OS/390 shell commands display information on your workstation screen, *standard output*. If you redirect the output, you can save the output from a command in a file instead. The output is sent to the file rather than to the screen. At the end of any command, enter:

```
>filename
```

For example:

```
cat file1 file2 file3 >outfile
```

writes the contents of the three files into another file called **outfile**. All the information in the original three files is concatenated into a single file, **outfile**.

When you redirect output with **>***filename* and it is an existing file, the output writes over any information that the file already contains. To *append* command output at the end of the file, use:

```
>>filename
```

instead. For example:

```
sort -u file1 >output 2>>outerr
```

redirects the result of the sort to the file named **output** (instead of standard output) and appends any error messages to the file **outerr**, which is a record of errors encountered during various sorts.

Suppose you entered:

```
sort -u filea 2>&1 >output
```

In this command, you see two redirections:

- Error output from the sort is redirected to standard output (`&1`), the display screen.
- The result of the sort is redirected to the file named **output**.

Here is another example with two redirections, sending both standard error and standard output to a file. This command produces the program **hello** and a listing with error messages in a file called **hello.list**:

```
c89 -o hello -V hello.c >hello.list 2>&1;
```

Redirecting Input from a File

You can redirect input in much the same way that you redirect output. A command that normally takes input from standard input can be redirected to take input from a file instead. For example, with this **mailx** command, you can send the file **power** to another user.

```
mailx DEEJ <power
```

The file **power** becomes input to **mailx**, rather than your input from the keyboard.

Redirecting Error Output to a File

You can redirect error output from the workstation screen to a file, using **2>**. (As you remember, `2` is the file descriptor for **stderr**.) For example:

```
sort -u filea 2>errfile
```

sorts **filea**, checking for unique output records. Any messages regarding duplicate records are redirected to a file named **errfile**.

If you want to append error output to an existing file, use **2>>**.

And if you do not care about seeing the error output, you can just redirect it to **/dev/null**, also known as the“bit bucket”. This is equivalent to discarding the error messages.

```
sort -u filea 2>/dev/null
```

Closing a File

The operating system has a limit on the number of streams to a file that a process can open. The KornShell closes a stream for you when a shell script ends.However, to conserve on the number of active file streams, you can close regular files when you are finished working with them in a shell script. To close a regular file, use either of the following:

```
exec n<&-
exec n>&-
```

where *n* can be file descriptors 3 through 9.

Similarly, you can close standard output, standard input, and standard error when you do not need them. For example, for an application that does not display anything, you may want to close standard output. Here is the command syntax for those files:

- `exec 0<&-` (close standard input)
- `exec 1>&-` (close standard output)
- `exec 2>&-` (close standard error)

Dumping Nontext Files to Standard Output

The **od** command can dump the contents of a file to *standard output,* your workstation screen, in several different formats.

```
od file
```

dumps a file in octal.

```
od -h file
```

dumps the file in hexadecimal. Either of these may be useful if you want to check the actual contents of a nontext file. Other dump formats are available.

Setting Up an Alias for a Command

After you have used the shell for a while, you will probably find that there are some commands that you use frequently. Rather than typing them over and over, you can set up an *alias* for these commands. An alias is a personalized name that stands for all or part of a command. You can create an alias by entering:

```
alias name="string"
```

in response to the shell's usual prompt for input. This is not a normal command; it is an instruction to the shell itself.

For example, suppose you have a hard time remembering that the **mv** command actually renames files. To make life easier for yourself, you could set up a simple alias by entering this on your command line:

```
alias renam="mv"
```

From this point onward in your session, whenever the shell sees the command **renam**, the **renam** is replaced with **mv**. The alias facility lets you create more usable commands.

Clearly, you could use an alias to save yourself some typing too. You could define **c** as an alias for **cat**. Then you would enter:

```
c file
```

to get the effect of:

```
cat file
```

Note: If you issue an **exec sh**, alias names are not exported. For information on how to put alias definitions in your login script pointed to by the **ENV** variable, see “Customizing Your Shell Environment: The ENV Variable” on page 42.

DBCS Recommendation: We recommend that you usesinglebyte characters when specifying an alias name, because the POSIX standard states that alias names must contain only characters in the POSIX portable character set.

Defining an Alias

If you will be using an alias frequently, put the **alias** command in your profile file (**$HOME/.profile**). When you issue the OMVS command or start a shell with **sh –L**, the shell reads the aliases from the file and sets them up immediately. That way, you do not have to type them in every time you start using the shell. See “Customizing Your .profile” on page 39 for more information about customizing your profile file.

To display all the currently defined aliases, you just enter:

```
alias
```

and the shell displays them. You will see a number of aliases that you did not set up. These are *predefined aliases* that the shell always creates.

When the shell replaces an alias, it checks to see if the result is another alias. The shell continues to check for and replace aliases until no aliases remain or the replacement would result in an infinite loop of alias expansion. For example, the shell defines the alias **functions** as follows:

```
alias functions="typeset -f"
```

Now, you might say to yourself, “Why do I need to type **functions** when I could just set up the alias **f**?” You could therefore enter:

```
alias f=functions
```

Then you enter:

```
f abc
```

the shell replaces **f** with **functions**, which the shell in turn replaces with:

```
"typeset -f"
```

Redefining an Alias for a Session

You can redefine an alias during a session, even if it is defined in your profile file. If you enter the command:

```
alias name="string"
```

during a session and *name* is already an alias, the shell forgets the old meaning and uses the new meaning from then on.

Setting Up an Alias for a Particular Version of a Command

If you tend to use a command with the same options every time, you may want to set up an alias for the command with those particular options. Let's take an example. The **grep** command searches through files and prints out lines that contain a requested string.For example:

```
grep hello file
```

displays all the lines of *file* that contain the string `hello`. Normally, **grep** distinguishes between uppercase and lowercase letters; this means, for example, that the search in the previous example does *not* display lines that contained `HELLO`, `Hello`, and so forth. If you want **grep** to ignore the case of letters as it searches, you must specify the **–i** option, as in:

```
grep -i hello file
```

This finds `hello`, `HELLO`, `Hello`, and so on.

If you think you prefer to use the **–i** version of **grep** most of the time, you can define the alias:

```
alias grep="grep -i"
```

From this point on, if you use the command:

```
grep string file
```

it is automatically converted to:

```
grep -i string file
```

and you get the case-insensitive version of the command **grep**.

As another example, the **rm** command to delete (remove)a file has an **–i** option that prompts you to confirm the deletion. The filename and a question mark are displayed. For example, if you entered `rm -i file1` and **file1** is in your working directory, you would see the prompt:

```
file1: ?
```

before the system actually removes the file. You then enter `y` (yes) or `n` (no) in response. If you like this extra bit of safety, you might define:

```
alias rm="rm -i"
```

After this, when you call **rm**, it automatically checks with you before deleting a file, just to make sure that you really want to delete it.

It may seem odd to define an alias that has the same name as a command that is used in the alias, but this is so common that the OS/390 shell checks specially for an alias of the same name, and does the correct thing.

If you find yourself using the same option every time you call a command, you might consider creating an appropriate alias so that the shell automatically adds the option. Of course, the best place to define this alias is in your **.profile** file; then the alias is set up every time you invoke the shell.

Using Alias Tracking

Alias tracking can reduce the time the shell spends searching your search path (specified with the **PATH** variable) for a command; it helps shell scripts run faster. A *tracked alias* is a shell-created alias that is the full pathname for a command. The

shell automatically tracks everything it finds in the default path for executables (**/bin**). For example, if you enter the **ps** command, the shell creates the alias:

```
ps="/bin/ps"
```

To use alias tracking for commands in other locations, enter the command:

```
set -o trackall
```

The first time you enter a command, the shell creates an alias that is the full pathname of the command. For example, if the user **marcw** entered the **hello** command and the shell tracked the command, it would create the alias:

```
hello="/u/marcw/bin"
```

Each time you enter a command, the shell uses its tracked alias, instead of searching the **PATH** for the command.

To list your tracked aliases, enter the command:

```
alias -t
```

To turn off alias tracking, enter the command:

```
set +o trackall
```

However, you cannot turn off the automatic tracking of commands found in **/bin**.

To remove tracked aliases, use:

```
alias -r
```

Turning Off an Alias

If you have set up an alias like the one previously described for **rm**, you may find that you *do not* want the alias to apply in some situations. For example, when you delete a huge number of files, you probably do not want **rm** to ask if it is okay to delete each one. In this situation, you have several options:

- Get rid of the alias entirely. The command:

  ```
  unalias rm
  ```

  gets rid of the **rm** alias for the session. After this, when you enter **rm**, you get the real **rm** command.
- Escape the alias. If you put a backslash in front of an alias, the shell uses the real command rather than the alias. For example:

  \rm *file*
- Specify the full pathname. For example:

  /bin/rm *file*

  tells the shell to run the program in **/bin/rm**. The shell does not perform alias substitution when you specify a command as a pathname.

These alternatives should help you get around options that you have automatically associated with a command.

Combining Commands

There are several simple ways you can combine several commands on a single command line.

- You can run a series of commands, one after the other:
  Using a semicolon (**;**)
  Using **&&** and **||**
- You can run more than one command concurrently:
  Using a pipe (**|**) or a filter with a pipe

  The output from the first command is piped to the next command as the first command is running.

Using a Semicolon (;)

The shell lets you enter several commands on the same command line. To do this, just use the semicolon character to separate the commands; for example:

```
cd mydir ; ls
```

Also, if you have defined the alias:

```
alias l="ls –l"
```

you can enter:

```
cd mydir ; l
```

since you can use aliases such as **l** after a semicolon.

Using && and ||

When stringing together more than two commands, you may want to control the running of the second command based on the outcome of the first command. You can use:

&& If the command that precedes `&&` completes successfully, the command following `&&` is run. Leave a space on either side of the `&&` operator: `command && command`.

|| If the command that precedes **||** fails, the command following **||** is run. Leave a space on either side of the **||** operator: `command || command`.

Using a Pipe

The output from one command can be *piped in* as input to the next command. Two or more commands linked by a pipe (|) are called a *pipeline*. A pipeline is written as:

```
command | command | ...
```

You enter the commands on the same line and separate them by the “or-bar” character |.

Many OS/390 shell commands are well suited to being used in a pipeline. For example, the **grep** command searches for a particular string in input from a file or standard input (the keyboard). A command such as:

```
history | grep "cp"
```

displays all the **cp** commands recorded among the 16 most recently recorded commands in your history file. The command:

```
ls –l | grep "Jan"
```

uses **ls** to obtain information on the contents of the working directory and uses **grep** to search through this information and display only the lines that contain the string `Jan`. The pipeline displays the files that were last changed in January.

A *filter* is a command that can read from standard input and write to standard output. A filter is often used within a pipeline. In the following example, **grep** is the filter:

```
ps -e | grep cc | wc -l
```

lists all your processes currently active in the system, pipes the output to **grep**, which searches for every instance of the string *cc*. The output from **grep** is then piped to **wc**, which counts every line in which the string *cc* occurs and sends the number of lines to standard output.

Using Substitution in Commands

Another shell feature that is useful for programmers is *command substitution*. When encountering a construct of the form:

```
$(command)
```

or:

```
`command`
```

in an input command line, the shell runs the given *command*. It then puts the output of the command, after converting newlines into spaces, back into the command line, replacing *command*, and runs the new command line. This is called *command substitution*.

You may find the `$( )` syntax easier to use for long command lines. However, the `` ` ` `` (backward apostrophes) syntax is more traditional and accepted on older UNIX shells.

As an example of how a programmer could use command substitution, consider a file called **srclist**, containing the following list of source code filenames: **alpha.c**, **beta.c**, and **gamma.c**. If you enter the command:

```
grep printf $(cat srclist)
```

the shell runs **cat** against the contents of **srclist**, and rewrites the original command line, so that this line appears as:

```
grep printf alpha.c beta.c gamma.c
```

This line is then run, with **grep** searching through the given files, displaying lines that contain the string `printf`. This type of construct quickly locates all references to a particular variable or function in the source code for a program.

Using the find Command in Command Substitution Constructs

The **find** command is useful in command substitution constructs. **find** displays the names of files that have specified characteristics. For example:

```
find dir1 –name "*.c"
```

finds all files in the directory **dir1** whose names match the wildcard pattern `*.c`. In other words, it finds all files in that directory with names having the **.c** suffix.

The command:

```
ls -l $(find dir1 –name "*.c")
```

finds all the **.c** files and then uses **ls** to display information about these files.

Complicating things further, you could enter

```
ls -l $(find dir1 –name "*.c") | grep -F "Nov"
```

This sets up a pipeline that displays **ls** information only for files that were last changed in November. (To be perfectly accurate, it also displays information on files that have the string `Nov` in their names, too.)

Another useful **find** option has the form:

```
find path –ctime number
```

This says that you want to find files that have changed in the last *number* of days. For example:

```
ls -l $(find dir –ctime 1)
```

displays **ls** information on all files that changed either yesterday or today.

On many UNIX and AIX systems, the **find** command prints out the filenames only if you specify the **–print** option. Thus, you would have to enter:

```
find dir –name "*.c" –print
```

to get the results just described. The OS/390 UNIX **find** command automatically prints its results without **–print**. However, if you have an existing shell script or compatibility with UNIX systems is important to you, you can use **–print**.

For more information on the **find** command, see *OS/390 UNIX System Services Command Reference*.

Characters That Have Special Meaning to the Shell

Certain characters have special meaning to the shell; these are often called *metacharacters*. If you enter a command that contains any of these characters, the shell often assumes that you are using the character in its special sense.

Characters Used with Commands

Character	Usage
|	Pipes the output from one command to a second command; separates commands in a *pipeline*.
||	Separates two commands. If the command preceding || fails, it runs the following command (Boolean OR operator).
&	Runs a command in the background, if placed at the end of a command line. Used in redirection, &0 represents standard input, &1 represents standard output, and &2 represents standard error.
&&	Separates two commands. If the command preceding && succeeds, it runs the following command (Boolean AND operator).

; Separates sequential commands; allows you to enter more than one command on the same line.

( ) Around a sequence of commands, groups those commands that are to run as a separate process in a subshell environment. The commands run in a separate execution environment: changes to variables, the working directory, open files, and so on, will not remain in effect after the last command finishes.

( ) is also used to group mathematical operations.

{ } Around a sequence of commands, groups those commands that are run in the current shell environment. Changes to variables, etc., will affect the current shell.

Both { and } are reserved words to the shell. To make it possible for the shell to recognize these symbols, you must enter a blank or <newline> after the {, and a semicolon or <newline> before the }.

Following a command in a shell script, indicates the beginning of a comment.

$ At the beginning of a string, indicates it is a variable name.

**** The backslash character turns off the special meaning of the character that follows it. For more information, see “Using a Special Character without Its Special Meaning” on page 79.

' ' A pair of single quotes turns off the special meaning of all characters within the quotes. For more information, see “Using a Special Character without Its Special Meaning” on page 79.

" " A pair of double quotes turns off the special meaning of the characters within the quotes, except for $, `, ", and \. See “Using a Special Character without Its Special Meaning” on page 79 for more information.

Characters Used in Filenames

Character
Usage

/ Separates the components of a file’s pathname.

~ (Tilde) symbolizes your home directory when used by itself. When used together with a user ID, ~ symbolizes that user’s home directory. For example:

```
~susanb/.profile
```

refers to user SUSANB’s **.profile** file.

You can also use the ~ to refer to your “previous” working directory; for example, the command

```
cd ~-
```

returns you to the directory you were previously working in.

. When used as a component of a pathname, indicates the working directory.

.. When used as a component of a pathname, indicates the parent directory.

? Used as a wildcard character that can match any one character, except a leading dot (**.**).

* Used as a wildcard character that can match a sequence of zero or more characters, except a leading dot (.).

Redirecting Input and Output

Character	Usage	Example
<	Redirects input to a specified file.	“Redirecting Input from a File” on page 70.
>	Redirects output to a specified file.	“Redirecting Command Output to a File” on page 69.
>>	Redirects output to be appended to the end of the specified file.	“Redirecting Command Output to a File” on page 69.
2>	Redirects error output to a specified file.	“Redirecting Error Output to a File” on page 70.

<<text	Reads standard input until it encounters *text*.	This is used in what is called a “here document.” Input is usually typed on the screen or in a shell script. For example, this script creates a file called **hello.c**, compiles it into **hello**, and then executes it:
		`echo "Creating program source..."`
		`if cat > hello.c <<End_Of_File`
		`main() {`
		`  puts("Hello, world!");`
		`}`
		`End_Of_File`
		`then`
		`  echo "Compiling program..."`
		`  if make hello`
		`  then`
		`  echo "Executing program..."`
		`  exec ./hello`
		`  else`
		`  exit $? # make failed`
		`  fi`
		`else`
		`  exit $? # cat failed`
		`  fi`
		
		When you run the shell script, it runs the **cat > hello.c** command using the input between the two End_of_File strings.

Using a Special Character without Its Special Meaning

If you do not want to use the special sense of the metacharacters, instruct the shell to ignore them by escaping them or quoting them. To do this, you use:

```
\
' '
" "
```

The Backslash (\)

The backslash character (\) turns off the special meaning of the character that follows it.For example:

```
echo it\'s me
```

prints:

```
it's me
```

If you just try:

```
echo it's me
```

without the backslash, the shell prints a > prompt after you press <Enter>instead of the usual $. The > prompt is a *continuation prompt*. An apostrophe ' without a backslash is taken to be the start of a string and the shell assumes that the string keeps going until you type another apostrophe, even if that goes on for several lines. The shell does not process the string until you type the closing apostrophe.

So remember to put a backslash in front of any special character, unless you know its special meaning and you want that meaning. Because a backslash itself is a special character, you must type two of them whenever you want a single backslash.

A Pair of Single Quotes (' ')

A pair of single quotes (' ') turns off the special meaning of *all characters* within the quotes.

A Pair of Double Quotes (" ")

A pair of double quotes (" ") turns off the special meaning of the characters within the quotes, except for $, `, ", and \.

Using a Wildcard Character to Specify Filenames

If you have used other operating systems, you are probably familiar with the concept of *wildcard characters*. (In an MVS context, the wildcard character is referred to as a *global character,* or *pattern-matching character*.) A wildcard character is a special character that may be used to save typing in filenames in shell commands. The OS/390 shell recognizes several different wildcard characters:

```
*
?
[ ]
```

The * Character

The asterisk (*) stands for any sequence of zero or more characters, except a leading dot. You can use the asterisk in filenames. For example:

```
ls aa*
```

lists all files in the working directory with names that begin with `aa`.

The command:

```
mv *.c dir1/dir2
```

moves every file with the **.c** suffix from your working directory to the directory **dir1/dir2**.

You can use the * wildcard character in directory names as well as in filenames. For example:

```
cat */*.c
```

displays the contents of all files that have the **.c** suffix, in directories under your working directory.

The ? Character

In a pathname, the question mark **?** can stand for any single character, except a leading dot. For example:

```
file.?
```

refers to any and all files with names that consist of **file.** followed by any single character. This can mean **file.a**, **file.b**, **file.c**, and so on ... whichever of the files currently exist.

You can combine ***** and **?**.

```
ls *.?
```

displays the names of all files under the working directory that have one-character filename suffixes.

Again, you can use the **?** in directory names as well as filenames. For example:

```
ls ???/*
```

shows all files in every directory under your working directory that have a three-character name.

The Square Brackets [ ]

Square brackets containing one or more characters stand for any one of the contained characters. For example:

```
[bch]at
```

matches **bat**, **cat**, or **hat**.

```
ls [abc]*
```

lists all files in the working directory the names of which start with `a`, `b`, or `c`, followed by any other sequence of zero or more characters. In other words, it lists all files whose names start with `a`, `b`, or `c`.

You can specify ranges of characters inside the square brackets by specifying the first character in the sequence, a hyphen (**-**), and the last character. For example:

```
[a–m]
```

This matches any character from `a` through `m`.

Suppose, for example, that you want to copy the contents of the working directory into two separate directories. You might enter:

```
cp [a–m]* dira
```

to copy all files with names beginning with the letters `a` through `m` to the directory **dira**, and then issue the second command:

```
cp [n–z]* dirb
```

to copy the rest of the files to the directory **dirb**. A command such as:

```
rm *.[a-z]
```

removes every file with a suffix consisting of a single lowercase letter.

If the first character inside a bracket construct is an exclamation mark **!**, the construct matches any character that *is not* inside the brackets. For example:

```
ls [!a–m]*
```

lists any file that *does not* begin with one of the letters in the range `a` through `m`.

In the same way:

```
rm [!0-9]*
```

removes any file with a name that does not start with a digit.

Retrieving Previously Entered Commands

In the shell, you can retrieve previously issued commands using:

- The **history** command, combined with the **r** command
- The two retrieve function keys that are part of the TSO/E OMVS command interface to the shell
- Command-line editing, when you are using an asynchronous terminal interface

Retrieving Commands from the History File

The shell records each command that you enter in a file under your *home directory*. This file is called the *history file*; its name is **.sh_history**.If you enter the command:

```
history
```

the shell displays the current contents of your history file. Each command is numbered.

You can rerun any of the commands in your history file by typing **r**, followed by a space, followed by the number of the command you want to use. Think of **r** as the "redo" command.

For example, suppose that you are a programmer and you enter a complicated command to compile part of a program. The program contains a syntax error, so you call a text editor to edit the source code and correct the problem. Now you want to run the same compile command on the corrected program. You may save yourself a good deal of typing by using:

```
history
```

to find out the number of the previous compile command; you can then run the command with **r**.

Another time-saver is to specify your shell prompt as:

```
PS1='(!)$'
```

in your **.profile**. The shell prompt is then preceded by the number assigned to the command in the command history file.

This is how you use the command numbers to enter a command. To repeat command number 14, enter:

```
r 14
```

The shell displays the original command 14 in the output area of the screen and then runs it. If you get another error, you can correct it, and then compile again with another**r 14**. You can perform the operation many times, but you have to type the original only once.

If you type **r** followed by a space, followed by a string of characters (not beginning with a digit), the shell checks backward through the history file and runs the most recent command that begins with the given string. For example, let's look at the compilation example. Suppose you are using the **c++** command to compile your program. Then:

```
r c++
```

looks back through the history and runs the most recent **c++** command. You do not even have to check on the number of the command you want to enter. The shell displays the selected command in the output area of the screen and then runs it.

This *backward-search* feature of **r** can search for aliases as well as normal commands. **r** searches for the beginning of the command line as you typed it, not the way that the line looked after the alias was replaced.

If you enter **r** without a number after it, the shell repeats the most recent command.

Editing Commands from the History File

Suppose that you have a sequence of source files named **file1.c**, **file2.c**, **file3.c**, and so on that you want to compile with similar **c89** commands. This situation is a little different from the one discussed in the previous section. You do not want to rerun the *same* command for each file; the command has the same form each time, but you have to specify in a new filename each time.

You can still do this using the history file. The command:

```
r old_string=new_string command
```

runs a previous *command* but replaces the first occurrence of the *old* string with the *new* string. For example, suppose you compile **file1.c** with:

```
c89 options file1.c
```

Then the command:

```
r file1=file2 c89
```

tells the shell to search back for the most recent **c89** command and to change **file1** to **file2**. The shell makes this change, and then displays and runs the modified command.

```
r file2=file3 c89
```

performs the same kind of operation, changing **file2** in the previous command to **file3** and then going ahead with the compilation. This saves you the trouble of retyping all the options for the command.

As mentioned earlier, entering `alias` displays all the currently defined aliases. You will see a number of aliases that you didn't set up; for example:

```
history="fc –l"
```

The **history** command is actually a *predefined* alias for the **fc** command with the **–l** option. The **fc** command is used to display and edit commands in the history file. Generally, it is easier to remember to type **history**, so the shell predefines this alias.

If you have displayed the predefined aliases, you probably noticed that **r** is also a predefined alias. It also stands for a version of the **fc** command. As with **history**, the **r** alias was created because it's easier to use and read than the straight **fc** command. For full details about **fc**, see *OS/390 UNIX System Services Command Reference*.

Using the Retrieve Function Keys

When you are using the OMVS interface, there are two function key settings for retrieving commands:

Retrieve
: This key performs a "backward retrieve" function. It retrieves a saved command from a stack of saved input lines, starting with the most recent and moving down to the oldest available line.

FwdRetr
: This key is used with the Retrieve key to retrieve commands from the stack of saved input lines. If you press the Retrieve key one too many times and go past the line you want, you can press the FwdRetr key to display the line that was previously retrieved by the Retrieve key.

Press the Retrieve key repeatedly until the command you want to use is displayed on the command line. Once the command is displayed, you can modify the command or use it as it is displayed. Press <Enter> to run the command.

Command-Line Editing

When you use **rlogin** or **telnet** to login to the shell, you can use command-line editing. Command-line editing lets you access commands from your history file, edit them, and run the result. You have already seen this process before, when reading about some of the features of the **r** command.

Command editing is useful at those times when you are running the same sequence of commands, or slight variations on the same sequence of commands. The point of command editing is to save yourself the trouble of typing the same thing over and over again—look especially for long commands that normally require a lot of typing. Command editing is also useful when you have made a mistake in typing a command line and wish to correct it.

Using the vi Command Editor

If you run the command:

```
set -o vi
```

or

```
export EDITOR=vi
```

it tells the shell that you want the ability to edit commands the way that you normally edit text with **vi**; you are set up for **vi** command editing. Whenever the shell prompts you for input, it is as if the shell puts you into **vi** insert mode on a new line at the end of the history file. You can type in a new command just as you normally would.

You can also press <Esc> to enter a **vi**-like command mode. When you enter command mode, you can use the usual cursor movement commands to move around on the command line, or to move up and down in the history file. For example:

- Press the **k** key to move back to the previous line in the history file (the last command line you entered). Press the **k** key again, and you move to the line before that.
- Press **j** and you move forward in the history file.

In this way it is simple to retrieve recent commands from the history file. You can then edit them using standard **vi** commands. For example, you can use **$** to move to the end of the line, and **A** to begin appending text to the end of the line. When you have edited the line to produce the command that you want to run, simply press <Enter> to run that line.

As you might expect, you can use these search commands:

```
/string
?string
```

to search backwards and forwards through the history file. You can edit the command line with these **vi** commands:

w Move to next word
b Move to previous word
d delete
c change
a append
i insert
u undo

and many of the other **vi** commands. For a complete list of available commands, see the description of **shedit** in *OS/390 UNIX System Services Command Reference* .

Using the emacs Command Editor

To set up for **emacs** command editing, enter:

```
set -o emacs
```

This lets you use commands identical to **emacs** commands to edit your shell command line. For a more information, see the description of **shedit** in *OS/390 UNIX System Services Command Reference*.

Using Record-Keeping Commands

Record-keeping commands can be very helpful for programmers. For example, suppose you have a program that is split into several source files. For the sake of simplicity, assume that the source files all have the extension **.c** and are all stored in a subdirectory called **src**. (To read about extensions, see “Naming Files” on page 213.)

It is often the case that you want to find out which source files in the subdirectory refer to a particular variable or function. You can do this very simply with the command:

```
grep 'name' src/*.c
```

The command checks all the appropriate files in the subdirectory **src** and displays the lines that contain the given *name*. Each line is labeled with the name of the file that contains the line. You can quickly find the use of a function or data object in source files.

As another example of using record-keeping commands, suppose that you are working on a large program and every few days you back up the source code for the program by copying it to a directory in a different file system (as a precaution). You would like to compare the current versions of your source files with one of the saved versions, to find out what changes have been made between the two. The command:

```
diff oldfile newfile
```

prints out all the differences between two versions of a file, making comparisons possible.

The **cksum** command gives a checksum for each file. If applied to two versions of what was at one time the same file, **cksum** gives a convenient way to tell if the files are still the same. It does not, however, indicate what the differences are.

The **find**command also has applications to programming. For example, suppose you are looking for a particular C source program but cannot remember where it is stored.

```
find / –name '*.c'
```

searches all the files and file systems, starting at the root, and displays the names of all files with the **.c** extension.

Finding Elements in a File and Presenting Them in a Specific Format

awk is a powerful command that can perform many different operations on files. The general purpose of **awk** is to read the contents of one or more files, obtain selected pieces of information from the files, and present the information in a specified format.

One simple way to use **awk** is with a command line with the form:

```
awk '/regexp/ {action}' file
```

This asks **awk** to obtain information from the specified file. **awk** obtains the information by performing the specified *action* on every line in the file that contains a string matching the given regular expression, *regexp*. (For further information, see the appendix on regular expressions in *OS/390 UNIX System Services Command Reference*.) For example:

```
awk '/abc/ {print}' file
```

displays every record in the file that contains the string `abc`.

For more discussion on using **awk**, see Appendix D.

Timing Programs

The **time** command lets you time programs to find out how much processor time they actually require. You might use this to compare two versions of a program to see if one runs faster than the other. You can run a program with:

```
time command-line
```

where *command-line* is a command line that invokes the program you want to time. **time** runs the program and displays:

- The total time the program took to execute, labeled `real`
- The total time spent in the user program, labeled `user`
- The central processor time spent performing system services for the user, labeled `sys`

Using the passwd Command

You can change user's passwords by using the **passwd** command:

passwd [**-u** *userid*]

The **passwd** command changes the login password for the user ID specified. If *userid* is omitted, the login name associated with the current terminal is used. You are prompted for the new password, which may be truncated to the length defined as the maximum length for the passwords.

For example:

```
passwd
```

changes the password for the invoker. The invoker is prompted for the old password and the new password values.

Non-superusers can change the password for another user if they know the user ID and current password. Another example changes the password for user ID Steve:

```
passwd -u steve
```

For more information about the **passwd** command, see *OS/390 UNIX System Services Command Reference*.

Switching to Superuser or Another ID

With the **su** command, you can switch to any user ID, including the superuser. A user can switch to superuser authority (with an effective UID of `0`), if the user is permitted to the BPX.SUPERUSER FACILITY class profile within the Resource Access Control Facility (RACF). Either the ISPF shell or the **su** shell command can be used for switching to superuser authority.

If you do not specify a user ID, the **su** command changes your authorization to that of the superuser. If you specify a user ID, **su** changes your authorization to that of the specified user ID.

When you switch to superuser (UID `0`) without specifying a userid, you keep your MVS identity (TSO/E ID). You keep your access authority to MVS data sets, while gaining authority to access any HFS files.

When you change user ID by specifying a user ID and password, you assume the MVS identity of the new userid even if the userid has UID `0`.

If you use the –s option on the **su** command you will not be prompted for a password. Use this option if you have access to the SURROGATE facility class profile BPX.SRV.*userid*. The *userid* is the MVS userid associated with the target UID.

To return to your own user ID, type:

```
exit
```

This returns you to the shell in which you entered the **su** command.

Using the whoami Command

The **whoami** command displays a username associated with the effective user ID, unlike the **who am i** command which displays the login name.

For example, if you login as 'user1' but then you use the **su** command to change to 'user2':

```
command                 returned
who am I                user1
whoami                  user2
```

For more information on the **whoami** command, see *OS/390 UNIX System Services Command Reference*.

Using the tso Command

To run a TSO/E command from the shell or in a shell script, simply preface the TSO/E command with the **tso** shell command; for example:

```
tso -t tso_command
```

There are two options you can use:

- Specify the *-t* option to run a command through the TSO/E service routine. The command output is written to **stdout**. If you specify a relative pathname, the command looks for the file in your current directory.

  Note: TSO/E has some restrictions on the type of commands that can be run using the TSO/E service routine (mini-TSO environment). In summary, you cannot run the following commands in this environment:
  - Commands that run authorized
  - FIB (foreground initiated background) commands
  - Other commands that require the TSO/E task structure, i.e., interactive commands such as **oedit**, where interactive means that the user can interact with the command processing while issuing additional terminal input (subcommands, function keys). For example, once the **oedit** command is entered, the user can enter additional subcommands to add more lines and then quit or exit the command.

  For a full description of the restrictions, see the section on IKJTSOEV in *OS/390 TSO/E Programming Guide*.
- Specify the *-o* option to run a TSO command as if it had been entered on the OMVS command line and run using the TSO subcommand or function key. If you use a relative pathname, the command looks for the file in the working directory of your TSO/E session, which is typically your home directory.

If no option is specified, the following rules are applied in this order:

1. If **stdout** is not a tty, the TSO service routine is used since it is possible that the command output is redirected to a file or piped to another command. Otherwise,
2. If the controlling tty supports 3270 passthrough mode, OMVS is used. Otherwise,
3. The TSO service routine is used.

The **tso** command supports several environment variables. For more information about the **tso** command and the environment variables associated with it, see *OS/390 UNIX System Services Command Reference*.

Online Help

Two help facilities are available with the shell:

- The **man** command, which you can use to display help information about a shell command. The man page is displayed in your shell session, and you can work in the shell while viewing the help information.
- The TSO/E OHELP command, which displays online reference information about shell commands, TSO/E commands, C functions, callable services, and messages issued by the shell and **dbx**.

  The IBM BookManager READ product is a requirementfor OHELP. The help information is displayed in a BookManager session; while viewing the help information, you cannot work in the shell.

Using the man Command

You can use the **man** command to get help information about a shell command. The **man** syntax is:

man *command_name*

- To scroll the information in a man page, press <Enter>.
- To end the display of a man page, type **q** and press <Enter>.

To search for a particular string in a system that has a list of one-line command descriptions, use the *-k* option:

man *-k string*

For example, to produce a list of all the shell commands for editing, you could type:

```
man -k edit
```

You can use the **man** command to view manual descriptions of TSO/E commands. To do this, you must prefix all commands with **tso**. For example, to view a description of the MOUNT command, you would enter:

```
man tsomount
```

You can also use the **man** command to view manual descriptions of **dbx** subcommands. To do this, you must prefix all subcommands with **dbx**. For example, to view a description of the **dbx alias** subcommand, you would enter:

```
man dbxalias
```

For complete information about the **man** command, see *OS/390 UNIX System Services Command Reference*.

Using the OHELP Command

The TSO/E OHELP command provides a similar capability to the **man** shell command. OHELP displays online reference information about commands, C functions, callable services, and messages issued by the shell and **dbx**. For information on entering TSO/E commands in TSO/E, the shell, and ISPF, see “Entering a TSO/E Command” on page 202.

Your system must have the BookManager READ productinstalled for you to use OHELP.

The OHELP syntax is:

```
OHELP ref_id search_item
```

ref_id A number that specifies the online book to be searched. The default is 1 for the *OS/390 UNIX System Services Command Reference*. Each installation can define which number is associated with each book. To see the list of available books and the number associated with each book, type `ohelp`.

search_item

This can specify a:

- Command name
- C function name
- Callable service name
- Message number
- Text string (enclosed in double quotes)

If you omit this operand, OHELP displays the table of contents of the book specified by the `ref_id`.

Example: Getting Help for a Command

For example, if you want information on the **cp** shell command, you would enter:

```
OHELP cp
```

(You do not need to enter the value 1 because 1 is the default.)

The output displayed looks similar to Figure 14:

```
                      List All Topics with Matches

Fuzzy matches for: CP
                                              Search matches 1 to 13 of 13
2.33     cp -- Copy a file
A.0      Appendix A.  OS/390 Shell Command Summary
2.21     chcp -- Set or query ASCII/EBCDIC code pages for the terminal
FRONT_1  Permuted Index
2.17     cat -- Concatenate or display a text file
2.89     ln -- Create a link to a file
2.107    mv -- Rename or move a file or directory
2.133    rm -- Remove a directory entry
2.159    touch -- Change the file access and modification times
2.162    trap -- Intercept abnormal conditions and interrupts
2.34     cpio -- Copy in/out file archives
2.42     dd -- Convert and copy a file
2.184    vi -- Use the display-oriented interactive text editor


Command ===> _____________________________________________ SCROLL ===> PAGE
 F1=Help     F4=Text     F5=No Text  F6=Review   F7=Bkwd     F8=Fwd
F10=Explain F12=Cancel
```

Figure 14. Sample Output from the Command OHELP cp

When you look at the output, you can see a boxed display overlaying another display. The boxed display, titled “List All Topics with Matches” lists all references to the **cp** command in the online *OS/390 UNIX System Services Command Reference*.

- `Fuzzy matches for: CP` is the heading for the list of references to **cp** that were found. BookManager converts the shell command name to uppercase.

- `Search Matches 1 to 13 of 13` indicates that this boxed display contains all of the search matches. If there were a very long list of search matches, you would need to scroll to the next screen to get to the end of the list.
- If you press PF4 (PF4=TEXT) while viewing the list, an explanation of the reason for the match is displayed.
- Your cursor is under the first item in the boxed display: `2.33 cp`. This is the **cp** command description from *OS/390 UNIX System Services Command Reference* . The first item in the list is usually the reference information for the language element you specified. Press <Enter>. You can read through the entire command description.
- To redisplay the boxed display of the search results, type `search cp`. Press <Enter>. Alternatively, you can position your cursor under the selection `Search` at the top of the screen. Press <Enter>. On the pulldown menu, select `List all topics with matches` and press <Enter>.
- After you select a match, you can use type `find cp` to move to the next match

If you press the Cancel function key, the boxed display disappears and you see the underlying information: the table of contents for the online *OS/390 UNIX System Services Command Reference*.

To exit the online help, use the Cancel and Exit function keys, as appropriate, from each panel.

Example: Searching Help for All Instances of a Language Element Name

If you want to look at the reference information for all types of language element with the name **chmod**, you enter the command:

```
ohelp * chmod
```

The output displayed would look similar to this:

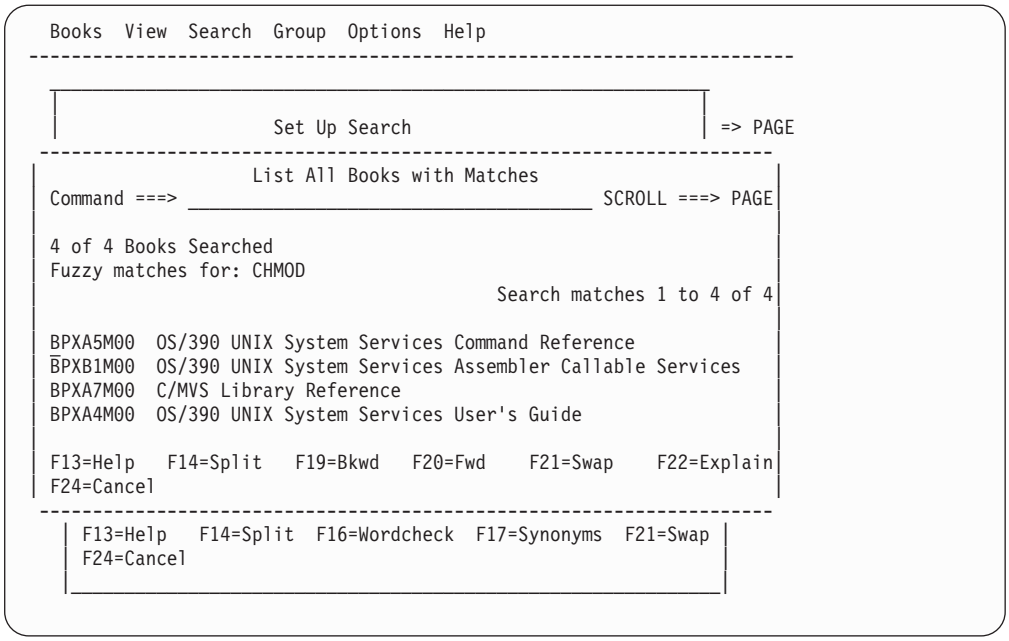


*Figure 15. Sample Output from the Command OHELP * chmod*

When you look at the output, you see a boxed display overlaying another display. The boxed display, titled “List All Books with Matches” lists all the reference books that document a language element named **chmod** command.

- `4 of 4 Books Searched` indicates that four books were searched.
- `Fuzzy matches for: CHMOD` is the heading for the list of references to **chmod** that were found. BookManager converts the shell command name to uppercase.
- `Search matches 1 to 4 of 4` indicates that this boxed display contains all of the search matches.
- Your cursor is under the first book in the list: `BPXA5M00`. If you press <Enter>, you see a boxed display similar to the one shown in Figure 14 on page 90, showing all search matches for **chmod** in the online *OS/390 UNIX System Services Command Reference*. The first item in the list is usually the reference information for the language element you specified. Press <Enter>. You can read through the entire function description.
- To return to the boxed display from the reference information, position your cursor under the selection `Search` at the top of the screen. Press <Enter>. On the pulldown menu, select `List all topics with matches` and press <Enter>.
- The remaining items listed are cross-references to the **chmod** function throughout the online *OS/390 UNIX System Services Command Reference*.

If you press <F12>, the boxed display disappears and you see the “Set Up a Search” panel, which allows you to search for a different name.

To exit the online help, use <F12> and <F3> as appropriate.

Searching for a Text String

To search for a text string, enclose the text in double quotes and specify the `ref_id` for the specific book you want to search. For example, the command

```
ohelp 4 "improper type"
```

will search the book *OS/390 UNIX System Services Messages and Codes* for messages that contain the text *improper type*.

If you are searching for a text string and you use an `*` for `ref_id`, OHELP will search all the books on the shelf and locate every instance of that string.

Shell Messages

Messages issued by the OS/390 shell and utilities are prefixed with the letters FSUM. To display online reference information about any shell message, use the OHELP command. The shell messages are documented in *OS/390 UNIX System Services Messages and Codes*.

Chapter 7. Working with tcsh Shell Commands

The shell is, above all, a *programmer's* interface. As a result, the shell commands are strongly slanted towards the needs of a programmer. The tcsh shell has many *general* tools that can help any programmer, and is specifically designed to have syntax similar to the C programming language. In addition, there are a number of commands designed especially for the C programmer.

Specifying Shell Command Options

Most of the commands discussed in this chapter accept options. Shell command options are usually specified by a minus sign (–) followed by a single character. For example, the **ls** command simply lists a directory's contents in multiple columns on your screen. However:

```
ls –F
```

distinguishes between various file types when listing the contents of a directory. (See "Listing Directory Contents" on page 209 for an example.)

```
ls –1
```

lists directory names in a single column.

Options consisting of a minus sign followed by a character are called *simple options*. You specify simple options after the name of the command and before any other arguments for the command (that is, arguments that are not options). For example, you would enter:

```
ls –1 dir1
```

to list the contents of **dir1** in a single column.

Command options and arguments must be typed as singlebyte characters. Additionally, delimiters such as a slash, curly brackets, and parentheses must be typed as singlebyte characters.

The order of options and arguments is important. If you enter:

```
ls dir1 –F
```

ls lists the contents of **dir1** and then tries to list the contents of the directory, or attributes of the file, called **–F**.

As a special notation, most tcsh shell commands let you specify a double minus sign (--) to separate the options from the nonoption arguments; -- means that there are no more options. Thus, if you really have a directory named **–F**, you could enter:

```
ls -- –F
```

to list the contents of that directory or the file attributes.

The tcsh shell gives you a shorthand way to specify more than one simple option to a command. For example, **–t** and **–v** are both simple options that you can specify with the **cat** command. (To find out what these options do, read the description of **cat** in *OS/390 UNIX System Services Command Reference* .) You could enter:

```
cat –t –v file
```

or you could combine the two options into:

```
cat –tv file
```

The order of the options is not important:

```
cat –vt file
```

is equivalent to the previous version of the command.

Specifying Options with Accompanying Arguments

In addition to simple options, some commands accept options that have accompanying arguments. Such options look like simple options followed by additional information. The argument may be a number, a string, the name of a file, or something else.

For example, if you read the description of **ps** in *OS/390 UNIX System Services Command Reference*, you will see that **ps** accepts an argument of the form:

```
–u userlist
```

When *OS/390 UNIX System Services Command Reference* shows part of a command line in *italics*, the italicized material is just a placeholder; when you actually use the command, you should fill in something else in its place. In this case, the *userlist* should be a string of one or more UID numbers or login names separated by commas and enclosed in single quotes. In the command:

```
ps –u 'macneil,wellie1'
```

the *userlist* string is macneil,wellie1. (If the string does not contain spaces, tabs, or other special characters, you can actually omit the enclosing single quotes, but the command is often easier to read if you use quotes anyway.) When executed, **ps** displays information for the specified users.

Help for Shell Command Usage

If you incorrectly specify a command, a usage note for the command is displayed. The usage note displays the proper format for the command. Often you can display a usage note deliberately if you specify the command with a *-?* option.

For online help information about a command, see “Online Help” on page 116.

Understanding Standard Input, Standard Output, and Standard Error

Once a command begins running, it has access to three files:

1. It reads from its *standard input* file. By default, standard input is the keyboard.
2. It writes to its *standard output* file.
   - If you invoke a shell command from the shell, a C program, or a REXX program invoked from TSO READY, standard output is directed to your terminal screen by default.
   - If you invoke a shell command, REXX program, or C program from the ISPF shell, standard output cannot be directed to your terminal screen. You can specify an HFS file or use the default, a temporary file.
3. It writes error messages to its *standard error* file.
   - If you invoke a shell command from the shell or from a C program or from a REXX program invoked from TSO READY, standard error is directed to your terminal screen by default.

- If you invoke a shell command, REXX program, or C program from the ISPF shell, standard error cannot be directed to your terminal screen. You can specify an HFS file or use the default, a temporary file.

  If the standard output or standard error file contains any data when the command completes, the file is displayed for you to browse.

Using the Shell:

In the shell, the names for these files are:
- **stdin** for the *standard input* file.
- **stdout** for the *standard output* file.
- **stderr** for the *standard error* file.

Using TSO/E:

When you are invoking the BPXBATCH utility, you can specify these standard files in MVS DD statements, TSO/E ALLOCATE commands, or DYNALLOC macros using the ddnames:
- STDIN for standard input
- STDOUT for standard output
- STDERR for standard error

For more information about BPXBATCH, see "The BPXBATCH Utility" on page 161.

Using ISPF:

When you run shell commands, REXX programs, and C programs from the ISPF shell, **stdout**, and **stderr** cannot be directed to your terminal. You can specify an HFS file, or use the default—a temporary file. If it has any contents, the file is displayed for you to browse when the command or program completes.

Redirecting Command Output to a File

Commands entered at the command line typically use the three standard files described in the previous section, but you can redirect the output for a command to a file you name. If you redirect output to a file that does not already exist, the system creates the file automatically.

Most shell commands display information on your workstation screen, *standard output*. If you redirect the output, you can save the output from a command in a file instead. The output is sent to the file rather than to the screen. At the end of any command, enter:

```
>filename
```

For example:

```
cat file1 file2 file3 >outfile
```

writes the contents of the three files into another file called **outfile**. All the information in the original three files is concatenated into a single file, **outfile**.

When you redirect output with **>***filename* and it is an existing file, the output writes over any information that the file already contains. To *append* command output at the end of the file, use:

```
>>filename
```

instead.

Another example:

```
(sort -u file1 >output) >&outerr
```

redirects the result of the sort to the file named **output** (instead of standard output) and redirects any error messages to the file **outerr**, which is a record of errors encountered during various sorts.

Suppose you entered:

```
sort -u filea >output
```

In this command, you see two redirections:

- Error output from the sort is redirected to standard output, the display screen.
- The result of the sort is redirected to the file named **output**.

Here is another example of redirection, sending both standard error and standard output to a file. This command produces the program **hello** and a listing with error messages in a file called **hello.list**:

```
c89 -o hello -V hello.c >&hello.list
```

Redirecting Input from a File

You can redirect input in much the same way that you redirect output. A command that normally takes input from standard input can be redirected to take input from a file instead. For example, with this **mailx** command, you can send the file **lessons** to another user.

```
mailx JAYD <lessons
```

The file **lessons** becomes input to **mailx**, rather than your input from the keyboard.

Redirecting Error Output to a File

You can redirect error output from the workstation screen to a file. For example:

```
(sort -u filea >dev/tty) >& outerr
```

sorts **filea**, checking for unique output records. Any messages regarding duplicate records are redirected to a file named **outerr**.

And if you do not care about seeing the error output, you can just redirect it to **/dev/null**, also known as the“bit bucket”. This is equivalent to discarding the error messages.

```
(sort -u filea >/dev/tty) >& /dev/null
```

Dumping Nontext Files to Standard Output

The **od** command can dump the contents of a file to *standard output,* your workstation screen, in several different formats.

```
od file
```

dumps a file in octal.

```
od -h file
```

dumps the file in hexadecimal. Either of these may be useful if you want to check the actual contents of a nontext file. Other dump formats are available.

Setting Up an Alias for a Command

After you have used the shell for a while, you will probably find that there are some commands that you use frequently. Rather than typing them over and over, you can set up an *alias* for these commands. An alias is a personalized name that stands for all or part of a command. You can create an alias by entering:

```
alias name "string"
```

in response to the shell's usual prompt for input. This is not a normal command; it is an instruction to the shell itself.

For example, suppose you have a hard time remembering that the **mv** command actually renames files. To make life easier for yourself, you could set up a simple alias by entering this on your command line:

```
alias renam "mv"
```

From this point onward in your session, whenever the shell sees the command **renam**, the **renam** is replaced with **mv**. The alias facility lets you create more usable commands.

Clearly, you could use an alias to save yourself some typing too. You could define **c** as an alias for **cat**. Then you would enter:

```
c file
```

to get the effect of:

```
cat file
```

Defining an Alias

If you will be using an alias frequently, put the **alias** command in your profile file (**$HOME/.tcshrc**). That way, you do not have to type them in every time you start using the shell. See "Understanding the Startup Files" on page 53 for more information about customizing your startup files.

To display all the currently defined aliases, you just enter:

```
alias
```

and the shell displays them.

Arguments in Aliases

Any arguments that follow an alias are treated just as if they had been following the command that the alias stands for. For example, if you define the alias **f** as follows:

```
alias f "ls"
```

the shell replaces **f** with **ls**, which is the command to list files in a directory.

You can refer to arguments in an alias by simply adding them at the end of the alias as you would with a command. For example:

```
f -la
```

would perform the **ls** command with the arguments **la**, which will list all the files in the directory in a long directory listing format. And,

```
f /bin
```

will list the contents of the **/bin** directory.

Redefining an Alias for a Session

You can redefine an alias during a session, even if it is defined in your profile file. If you enter the command:

```
alias name "string"
```

during a session and *name* is already an alias, the shell forgets the old meaning and uses the new meaning from then on.

Setting Up an Alias for a Particular Version of a Command

If you tend to use a command with the same options every time, you may want to set up an alias for the command with those particular options. Let's take an example. The **grep** command searches through files and prints out lines that contain a requested string.For example:

```
grep hello file
```

displays all the lines of *file* that contain the string `hello`. Normally, **grep** distinguishes between uppercase and lowercase letters; this means, for example, that the search in the previous example does *not* display lines that contained `HELLO`, `Hello`, and so forth. If you want **grep** to ignore the case of letters as it searches, you must specify the **–i** option, as in:

```
grep -i hello file
```

This finds `hello`, `HELLO`, `Hello`, and so on.

If you think you prefer to use the **–i** version of **grep** most of the time, you can define the alias:

```
alias grep "grep -i"
```

From this point on, if you use the command:

```
grep string file
```

it is automatically converted to:

```
grep -i string file
```

and you get the case-insensitive version of the command **grep**.

As another example, the **rm** command to delete (remove)a file has an **–i** option that prompts you to confirm the deletion. The filename and a question mark are displayed. For example, if you entered `rm -i file1` and **file1** is in your working directory, you would see the prompt:

```
file1: ?
```

before the system actually removes the file. You then enter `y` (yes) or `n` (no) in response. If you like this extra bit of safety, you might define:

```
alias rm "rm -i"
```

After this, when you call **rm**, it automatically checks with you before deleting a file, just to make sure that you really want to delete it.

It may seem odd to define an alias that has the same name as a command that is used in the alias, but this is so common that the shell checks specially for an alias of the same name, and does the correct thing.

If you find yourself using the same option every time you call a command, you might consider creating an appropriate alias so that the shell automatically adds the option. Of course, the best place to define this alias is in your **.tcshrc** file; then the alias is set up every time you invoke the shell.

Turning Off an Alias

If you have set up an alias like the one previously described for **rm**, you may find that you *do not* want the alias to apply in some situations. For example, when you delete a huge number of files, you probably do not want **rm** to ask if it is okay to delete each one. In this situation, you have several options:

- Get rid of the alias entirely. The command:

  ```
  unalias rm
  ```

  gets rid of the **rm** alias for the session. After this, when you enter **rm**, you get the real **rm** command.
- Escape the alias. If you put a backslash in front of an alias, the shell uses the real command rather than the alias. For example:

  ```
  \rm file
  ```
- Specify the full pathname. For example:

  ```
  /bin/rm file
  ```

  tells the shell to run the program in **/bin/rm**. The shell does not perform alias substitution when you specify a command as a pathname.

These alternatives should help you get around options that you have automatically associated with a command.

Combining Commands

There are several simple ways you can combine several commands on a single command line.

- You can run a series of commands, one after the other:
  Using a semicolon (**;**)
  Using **&&** and **||**
- You can run more than one command concurrently:
  Using a pipe (**|**) or a filter with a pipe

  The output from the first command is piped to the next command as the first command is running.

Using a Semicolon (;)

The shell lets you enter several commands on the same command line. To do this, just use the semicolon character to separate the commands; for example:

```
cd mydir ; ls
```

Also, if you have defined the alias:

```
alias l "ls –l"
```

you can enter:

```
cd mydir ; l
```

since you can use aliases such as **l** after a semicolon.

Using && and ||

When stringing together more than two commands, you may want to control the running of the second command based on the outcome of the first command. You can use:

&& If the command that precedes && completes successfully, the command following && is run. Leave a space on either side of the && operator: `command && command`.

|| If the command that precedes || fails, the command following || is run. Leave a space on either side of the || operator: `command || command`.

Using a Pipe

The output from one command can be *piped in* as input to the next command. Two or more commands linked by a pipe (|) are called a *pipeline*. A pipeline is written as:

command | *command* | ...

You enter the commands on the same line and separate them by the "or-bar" character |.

Many commands are well suited to being used in a pipeline. For example, the **grep** command searches for a particular string in input from a file or standard input (the keyboard). A command such as:

```
history | grep "cp"
```

displays all the **cp** commands recorded among the 16 most recently recorded commands in your history file. The command:

```
ls –l | grep "Jan"
```

uses **ls** to obtain information on the contents of the working directory and uses **grep** to search through this information and display only the lines that contain the string `Jan`. The pipeline displays the files that were last changed in January.

A *filter* is a command that can read from standard input and write to standard output. A filter is often used within a pipeline. In the following example, **grep** is the filter:

```
ps -e | grep cc | wc -l
```

lists all your processes currently active in the system, pipes the output to **grep**, which searches for every instance of the string *cc*. The output from **grep** is then piped to **wc**, which counts every line in which the string *cc* occurs and sends the number of lines to standard output.

Using Substitution in Commands

Another shell feature that is useful for programmers is *command substitution*. When encountering a construct of the form:

`` `command` ``

in an input command line, the shell runs the given *command*. It then puts the output of the command, after converting newlines into spaces, back into the command line, replacing *command*, and runs the new command line. This is called *command substitution*.

As an example of how a programmer could use command substitution, consider a file called **srclist**, containing the following list of source code filenames: **alpha.c**, **beta.c**, and **gamma.c**. If you enter the command:

```
grep printf `cat srclist`
```

the shell runs **cat** against the contents of **srclist**, and rewrites the original command line, so that this line appears as:

```
grep printf alpha.c beta.c gamma.c
```

This line is then run, with **grep** searching through the given files, displaying lines that contain the string `printf`. This type of construct quickly locates all references to a particular variable or function in the source code for a program.

Using the find Command in Command Substitution Constructs

The **find** command is useful in command substitution constructs. **find** displays the names of files that have specified characteristics. For example:

```
find dir1 –name "*.c"
```

finds all files in the directory **dir1** whose names match the wildcard pattern `*.c`. In other words, it finds all files in that directory with names having the **.c** suffix.

The command:

```
ls -l `find dir1 –name "*.c"`
```

finds all the **.c** files and then uses **ls** to display information about these files.

Complicating things further, you could enter

```
ls -l `find dir1 –name "*.c"` | grep -F "Nov"
```

This sets up a pipeline that displays **ls** information only for files that were last changed in November. (To be perfectly accurate, it also displays information on files that have the string `Nov` in their names, too.)

Another useful **find** option has the form:

```
find path –ctime number
```

This says that you want to find files that have changed in the last *number* of days. For example:

```
ls -l `find dir –ctime 1`
```

displays **ls** information on all files that changed either yesterday or today.

On many UNIX and AIX systems, the **find** command prints out the filenames only if you specify the **–print** option. Thus, you would have to enter:

```
find dir –name "*.c" –print
```

to get the results just described. The OS/390 UNIX **find** command automatically prints its results without **–print**. However, if you have an existing shell script or compatibility with UNIX systems is important to you, you can use **–print**.

For more information on the **find** command, see *OS/390 UNIX System Services Command Reference*.

Characters That Have Special Meaning to the Shell

Certain characters have special meaning to the shell; these are often called *metacharacters*. If you enter a command that contains any of these characters, the shell often assumes that you are using the character in its special sense.

Characters Used with Commands

Character	Usage
|	Pipes the output from one command to a second command; separates commands in a *pipeline*.
||	Separates two commands. If the command preceding || fails, it runs the following command (Boolean OR operator).
>	Redirects stdout.
<	Redirects stdin.
&	Runs a command in the background, if placed at the end of a command line.
>&	Used for redirecting stdout and stderr.
&&	Separates two commands. If the command preceding && succeeds, it runs the following command (Boolean AND operator).
;	Separates sequential commands; allows you to enter more than one command on the same line.
( )	Around a sequence of commands, groups those commands that are to run as a separate process in a subshell environment. The commands run in a separate execution environment: changes to variables, the working directory, open files, and so on, will not remain in effect after the last command finishes. ( ) is also used to group mathematical operations.
{ }	Around a sequence of commands, groups those commands that are run in the current shell environment. Changes to variables will affect the current shell. Both { and } are reserved words to the shell. To make it possible for the shell to recognize these symbols, you must enter a blank or <newline> after the {, and a semicolon or <newline> before the }.
#	Following a command in a shell script, indicates the beginning of a comment.
$	At the beginning of a string, indicates it is a variable name.
\	In general, the backslash character turns off the special meaning of the character that follows it. For more information, see “Using a Special Character without Its Special Meaning” on page 106.
' '	A pair of single quotes turns off the special meaning of all characters within the quotes. For more information, see “Using a Special Character without Its Special Meaning” on page 106.

" " A pair of double quotes turns off the special meaning of the characters within the quotes, except that `!event`, `$var`, and `` `cmd` `` will show history, variable, and command substitution. See “Using a Special Character without Its Special Meaning” on page 106 for more information.

Characters Used in Filenames

Character
Usage

/ Separates the components of a file’s pathname.

~ (Tilde) symbolizes your home directory when used by itself. When used together with a user ID, ~ symbolizes that user’s home directory. For example:

```
~valerie/.tcshrc
```

refers to user VALERIE’s **.tcshrc** file.

. When used as a component of a pathname, indicates the working directory.

.. When used as a component of a pathname, indicates the parent directory.

? Used as a wildcard character that can match any one character, except a leading dot (**.**).

***** Used as a wildcard character that can match a sequence of zero or more characters, except a leading dot (**.**).

Redirecting Input and Output

Character	Usage	Example
<	Redirects input to a specified file.	“Redirecting Input from a File” on page 98.
>	Redirects output to a specified file.	“Redirecting Command Output to a File” on page 97.
>>	Redirects output to be appended to the end of the specified file.	“Redirecting Command Output to a File” on page 97.
>&	Redirects stdout and stderr.	“Redirecting Error Output to a File” on page 98.

Character	Usage	Example

<<text	Reads standard input until it encounters *text*.	This is used in what is called a “here document.” Input is usually typed on the screen or in a shell script. For example, this script creates a file called **hello.c**, compiles it into **hello**, and then executes it:
		`# create program`
		`cat > hello.c << EOF`
		`main() {`
		`  puts("Hello, World!\n");`
		`}`
		`EOF`
		`# compile program`
		`c89 -o hello hello.c`
		`#execute program`
		`hello`
		
		When you run the shell script, it runs the **cat > hello.c** command using the input between the two End_of_File strings.

Using a Special Character without Its Special Meaning

If you do not want to use the special sense of the metacharacters, instruct the shell to ignore them by escaping them or quoting them. To do this, you use:

```
\
' '
" "
```

The Backslash (\)

The backslash character (\) turns off the special meaning of the character that follows it.For example:

```
echo it\'s me
```

prints:

```
it's me
```

If you just try:

```
echo it's me
```

without the backslash, the shell prints a > prompt after you press <Enter>instead of the usual $. The > prompt is a *continuation prompt*. An apostrophe ' without a backslash is taken to be the start of a string and the shell assumes that the string keeps going until you type another apostrophe, even if that goes on for several lines. The shell does not process the string until you type the closing apostrophe.

So remember to put a backslash in front of any special character, unless you know its special meaning and you want that meaning. Because a backslash itself is a special character, you must type two of them whenever you want a single backslash.

A Pair of Single Quotes (' ')

A pair of single quotes (' ') turns off the special meaning of *all characters* within the quotes.

A Pair of Double Quotes (" ")

A pair of double quotes turns off the special meaning of the characters within the quotes, except that `!event`, `$var`, and `` `cmd` `` will show history, variable, and command substitution.

Using a Wildcard Character to Specify Filenames

If you have used other operating systems, you are probably familiar with the concept of *wildcard characters*. (In an MVS context, the wildcard character is referred to as a *global character,* or *pattern-matching character*.) A wildcard character is a special character that may be used to save typing in filenames in shell commands. The tcsh shell recognizes several different wildcard characters:

```
*
?
[]
```

The * Character

The asterisk (*****) stands for any sequence of zero or more characters, except a leading dot. You can use the asterisk in filenames. For example:

```
ls aa*
```

lists all files in the working directory with names that begin with `aa`.

The command:

```
mv *.c dir1/dir2
```

moves every file with the **.c** suffix from your working directory to the directory **dir1/dir2**.

You can use the ***** wildcard character in directory names as well as in filenames. For example:

```
cat */*.c
```

displays the contents of all files that have the **.c** suffix, in directories under your working directory.

The ? Character

In a pathname, the question mark **?** can stand for any single character, except a leading dot. For example:

```
file.?
```

refers to any and all files with names that consist of **file.** followed by any single character. This can mean **file.a**, **file.b**, **file.c**, and so on ... whichever of the files currently exist.

You can combine ***** and **?**.

```
ls *.?
```

displays the names of all files under the working directory that have one-character filename suffixes.

Again, you can use the **?** in directory names as well as filenames. For example:

```
ls ???/*
```

shows all files in every directory under your working directory that have a three-character name.

The Square Brackets [ ]

Square brackets containing one or more characters stand for any one of the contained characters. For example:

```
[bch]at
```

matches **bat**, **cat**, or **hat**.

```
ls [abc]*
```

lists all files in the working directory the names of which start with `a`, `b`, or `c`, followed by any other sequence of zero or more characters. In other words, it lists all files whose names start with `a`, `b`, or `c`.

You can specify ranges of characters inside the square brackets by specifying the first character in the sequence, a hyphen (**-**), and the last character. For example:

```
[a–m]
```

This matches any character from `a` through `m`.

Suppose, for example, that you want to copy the contents of the working directory into two separate directories. You might enter:

```
cp [a–m]* dira
```

to copy all files with names beginning with the letters `a` through `m` to the directory **dira**, and then issue the second command:

```
cp [n–z]* dirb
```

to copy the rest of the files to the directory **dirb**. A command such as:

```
rm *.[a-z]
```

removes every file with a suffix consisting of a single lowercase letter.

If the first character inside a bracket construct is an exclamation mark **!**, the construct matches any character that *is not* inside the brackets. For example:

```
ls [!a–m]*
```

lists any file that *does not* begin with one of the letters in the range `a` through `m`.

In the same way:

```
rm [!0-9]*
```

removes any file with a name that does not start with a digit.

Retrieving Previously Entered Commands

In the tcsh shell, you can retrieve previously issued commands using:

- The **history** command, combined with the **!** command
- The two retrieve function keys that are part of the TSO/E OMVS command interface to the shell
- Command-line editing, when you are using an asynchronous terminal interface

Retrieving Commands from the History File

The shell records each command that you enter in a file under your *home directory*. This file is called the *history file*; its name is **.history**.If you enter the command:

```
history
```

the shell displays the current contents of your history file. Each command is numbered.

You can rerun any of the commands in your history file by typing **!**, followed by a space, followed by the number of the command you want to use.

For example, suppose that you are a programmer and you enter a complicated command to compile part of a program. The program contains a syntax error, so you call a text editor to edit the source code and correct the problem. Now you want to run the same compile command on the corrected program. You may save yourself a good deal of typing by using:

```
history
```

to find out the number of the previous compile command and then running the command with **!**. For example, if the history file shows you that the command you want to run is number 44, you would type:

```
! 44
```

to run the previous compile command.

Another time-saver is to specify your shell prompt as:

```
set prompt="\!>
```

in your **.tcshrc** file. The shell prompt is then preceded by the number assigned to the command in the command history file.

If you type **!** followed by a space, followed by a string of characters (not beginning with a digit), the shell checks backward through the history file and runs the most recent command that begins with the given string. For instance, look at the compilation example. Suppose you are using the **c++** command to compile your program. Then:

```
! c++
```

looks back through the history and runs the most recent **c++** command. You do not even have to check on the number of the command you want to enter. The shell displays the selected command in the output area of the screen and then runs it.

This *backward-search* feature of **!** can search for aliases as well as normal commands. **!** searches for the beginning of the command line as you typed it, not the way that the line looked after the alias was replaced.

If you enter **!!** without a number after it, the shell repeats the most recent command.

Editing Commands from the History File

Suppose that you have a sequence of source files named **file1.c**, **file2.c**, **file3.c**, and so on that you want to compile with similar **c89** commands. This situation is a little different from the one discussed in the previous section. You do not want to rerun the *same* command for each file; the command has the same form each time, but you have to specify in a new filename each time.

You can still do this using the history file. The command:

```
^old_string^new_string
```

runs a previous *command* but replaces the first occurrence of the *old* string with the *new* string. For example, suppose you compile **file1.c** with:

```
c89 options file1.c
```

Then the command:

```
^file1^file2
```

tells the shell to look at the previous command and to change **file1** to **file2**. The shell makes this change, and then displays and runs the modified command.

```
^file2^file3
```

performs the same kind of operation, changing **file2** in the previous command to **file3** and then going ahead with the compilation. This saves you the trouble of retyping all the options for the command.

Using the Retrieve Function Keys

If you are using the OMVS interface, there are two function key settings for retrieving commands:

Retrieve
: This key performs a “backward retrieve” function. It retrieves a saved command from a stack of saved input lines, starting with the most recent and moving down to the oldest available line.

FwdRetr
: This key is used with the Retrieve key to retrieve commands from the stack of saved input lines. If you press the Retrieve key one too many times and go past the line you want, you can press the FwdRetr key to display the line that was previously retrieved by the Retrieve key.

Press the Retrieve key repeatedly until the command you want to use is displayed on the command line. Once the command is displayed, you can modify the command or use it as it is displayed. Press <Enter> to run the command.

Command-Line Editing

When you use **rlogin** or **telnet** to login to the shell, you can use command-line editing. Command-line editing lets you access commands from your history file, edit them, and run the result. You have already seen this process before, when reading about some of the features of the **!** command.

Command editing is useful at those times when you are running the same sequence of commands, or slight variations on the same sequence of commands. The point of command editing is to save yourself the trouble of typing the same thing over and over again—look especially for long commands that normally require a lot of typing. Command editing is also useful when you have made a mistake in typing a command line and wish to correct it.

Using the vi Command Editor

If you run the command:

```
bindkey -v
```

it tells the shell that you want the ability to edit commands the way that you normally edit text with **vi**; you are set up for **vi** command editing. Whenever the shell prompts you for input, it is as if the shell puts you into **vi** insert mode on a new line at the end of the history file. You can type in a new command just as you normally would.

You can also press <Esc> to enter a **vi**-like command mode. When you enter command mode, you can use the usual cursor movement commands to move around on the command line, or to move up and down in the history file. For example:

- Press the **k** key to move back to the previous line in the history file (the last command line you entered). Press the **k** key again, and you move to the line before that.
- Press **j** and you move forward in the history file.

In this way it is simple to retrieve recent commands from the history file. You can then edit them using standard **vi** commands. For example, you can use **$** to move to the end of the line, and **A** to begin appending text to the end of the line. When you have edited the line to produce the command that you want to run, simply press <Enter> to run that line.

As you might expect, you can use these search commands:

```
/string
?string
```

to search backwards and forwards through the history file. You can edit the command line with these **vi** commands:

w Move to next word
b Move to previous word
d delete
c change
a append
i insert
u undo

and many of the other **vi** commands. For a complete list of available commands, see the description of **tcsh** in *OS/390 UNIX System Services Command Reference* .

Using the emacs Command Editor

To set up for **emacs** command editing, enter:

```
bindkey -e
```

This lets you use commands identical to **emacs** commands to edit your shell command line. For a more information, see the description of **tcsh** in *OS/390 UNIX System Services Command Reference*.

Using Filename Completion

Note: Filename Completion requires the use of the TAB key. This key must be mapped correctly for the feature to work. Most connections through **telnet** and **rlogin** will transmit the TAB information correctly. If you are connected in any other manner, this feature may not work correctly.

The tcsh shell provides a time saving feature for completing filenames. Rather than having to type out the entire string to access a file or execute a program, you can type just the first letter or letters and let the shell help you with the rest.

For example, if you have a file called *phonebook*, and you want to list the contents of this file on the screen with the **more** command, you can do so by typing the command, the first letter or letters of the file, and then pressing the TAB key. For example, if you type:

```
more ph
```

and then press the TAB key, the shell will provide you with:

```
more phonebook
```

you can then press ENTER and execute the command.

If you have more than one filename that matches the letter or letters you have typed, the shell will alert you with a beep. For example, if you have three files, called *list1*, *list2*, and *list3*, and you type:

```
more li
```

and press TAB, the beep will sound, and the shell will complete the filename as far as it can:

```
more list
```

you must then type *1*, *2*, or *3* and press ENTER.

If you are unsure of how many files there are, or which one you want, you can type <CRTL-D> when the shell beeps, and you will be provided with matching names. For example:

```
> more list
list1   list2  list3
> more list
```

Underneath the matching names the command prompt is displayed again. Now you can enter the number that you wish and then press ENTER.

If there are no matches for the letter or letters you have typed, the shell will beep, but when you press <CRTL-D>, nothing will be displayed.

You can also use filename completion to aid in changing between directories with long paths. If you keep files in the directory *stuff/data/graphics*, it is easier to use filename completion to access the directory than to type the entire path by hand. For example, if you are in your home directory, and *stuff* is a subdirectory containing *data/graphics*, and you want to change into that directory, you can do the following:

```
cd s    [TAB]
cd stuff/
cd stuff/d    [TAB]
cd stuff/data
cd stuff/data/g    [TAB]
cd stuff/data/graphics
```

then press ENTER, and the directory change command will execute.

More information on filename completion and the **complete** command can be found in *OS/390 UNIX System Services Command Reference*.

Using Record-Keeping Commands

Record-keeping commands can be very helpful for programmers. For example, suppose you have a program that is split into several source files. For the sake of simplicity, assume that the source files all have the extension **.c** and are all stored in a subdirectory called **src**. (To read about extensions, see “Naming Files” on page 213.)

It is often the case that you want to find out which source files in the subdirectory refer to a particular variable or function. You can do this very simply with the command:

```
grep 'name' src/*.c
```

The command checks all the appropriate files in the subdirectory **src** and displays the lines that contain the given *name*. Each line is labeled with the name of the file that contains the line. You can quickly find the use of a function or data object in source files.

As another example of using record-keeping commands, suppose that you are working on a large program and every few days you back up the source code for the program by copying it to a directory in a different file system (as a precaution). You would like to compare the current versions of your source files with one of the saved versions, to find out what changes have been made between the two. The command:

```
diff oldfile newfile
```

prints out all the differences between two versions of a file, making comparisons possible.

The **cksum** command gives a checksum for each file. If applied to two versions of what was at one time the same file, **cksum** gives a convenient way to tell if the files are still the same. It does not, however, indicate what the differences are.

The **find**command also has applications to programming. For example, suppose you are looking for a particular C source program but cannot remember where it is stored.

```
find / –name '*.c'
```

searches all the files and file systems, starting at the root, and displays the names of all files with the **.c** extension.

Finding Elements in a File and Presenting Them in a Specific Format

awk is a powerful command that can perform many different operations on files. The general purpose of **awk** is to read the contents of one or more files, obtain selected pieces of information from the files, and present the information in a specified format.

One simple way to use **awk** is with a command line with the form:

```
awk '/regexp/ {action}' file
```

This asks **awk** to obtain information from the specified file. **awk** obtains the information by performing the specified *action* on every line in the file that contains

a string matching the given regular expression, *regexp*. (For further information, see the appendix on regular expressions in *OS/390 UNIX System Services Command Reference*.) For example:

```
awk '/abc/ {print}' file
```

displays every record in the file that contains the string `abc`.

For more discussion on using **awk**, see Appendix D.

Timing Programs

The **time** command lets you time programs to find out how much processor time they actually require. You might use this to compare two versions of a program to see if one runs faster than the other. You can run a program with:

```
time command-line
```

where *command-line* is a command line that invokes the program you want to time. **time** runs the program and displays:

- The total time the program took to execute, labeled `real`
- The total time spent in the user program, labeled `user`
- The central processor time spent performing system services for the user, labeled `sys`

Using the passwd Command

You can change user's passwords by using the **passwd** command:

```
passwd [-u userid]
```

The **passwd** command changes the login password for the user ID specified. If *userid* is omitted, the login name associated with the current terminal is used. You are prompted for the new password, which may be truncated to the length defined as the maximum length for the passwords.

For example:

```
passwd
```

changes the password for the invoker. The invoker is prompted for the old password and the new password values.

Non-superusers can change the password for another user if they know the user ID and current password. Another example changes the password for user ID Bonnie:

```
passwd -u bonnie
```

For more information about the **passwd** command, see *OS/390 UNIX System Services Command Reference*.

Switching to Superuser or Another ID

With the **su** command, you can switch to any user ID, including the superuser. A user can switch to superuser authority (with an effective UID of `0`), if the user is permitted to the BPX.SUPERUSER FACILITY class profile within the Resource Access Control Facility (RACF). Either the ISPF shell or the **su** shell command can be used for switching to superuser authority.

If you do not specify a user ID, the **su** command changes your authorization to that of the superuser. If you specify a user ID, **su** changes your authorization to that of the specified user ID.

When you switch to superuser (UID `0`) without specifying a userid, you keep your MVS identity (TSO/E ID). You keep your access authority to MVS data sets, while gaining authority to access any HFS files.

When you change user ID by specifying a user ID and password, you assume the MVS identity of the new userid even if the userid has UID `0`.

If you use the –s option on the **su** command you will not be prompted for a password. Use this option if you have access to the SURROGATE facility class profile BPX.SRV.*userid*. The *userid* is the MVS userid associated with the target UID.

To return to your own user ID, type:

```
exit
```

This returns you to the shell in which you entered the **su** command.

Using the whoami Command

The **whoami** command displays a username associated with the effective user ID, unlike the **who am i** command which displays the login name.

For example, if you login as 'user1' but then you use the **su** command to change to 'user2':

command	**returned**
`who am I`	`user1`
`whoami`	`user2`

For more information on the **whoami** command, see *OS/390 UNIX System Services Command Reference*.

Using the tso Command

To run a TSO/E command from the shell or in a shell script, simply preface the TSO/E command with the **tso** shell command; for example:

```
tso -t tso_command
```

There are two options you can use:

- Specify the *-t* option to run a command through the TSO/E service routine. The command output is written to **stdout**. If you specify a relative pathname, the command looks for the file in your current directory.

  Note: TSO/E has some restrictions on the type of commands that can be run using the TSO/E service routine (mini-TSO environment). In summary, you cannot run the following commands in this environment:
  - Commands that run authorized
  - FIB (foreground initiated background) commands
  - Other commands that require the TSO/E task structure, i.e., interactive commands such as **oedit**, where interactive means that the user can interact with the command processing while issuing additional terminal input (subcommands, function keys). For example, once the **oedit**

command is entered, the user can enter additional subcommands to add more lines and then quit or exit the command.

For a full description of the restrictions, see the section on IKJTSOEV in *OS/390 TSO/E Programming Guide*.

- Specify the *-o* option to run a TSO command as if it had been entered on the OMVS command line and run using the TSO subcommand or function key. If you use a relative pathname, the command looks for the file in the working directory of your TSO/E session, which is typically your home directory.

If no option is specified, the following rules are applied in this order:

1. If **stdout** is not a tty, the TSO service routine is used since it is possible that the command output is redirected to a file or piped to another command. Otherwise,
2. If the controlling tty supports 3270 passthrough mode, OMVS is used. Otherwise,
3. The TSO service routine is used.

The **tso** command supports several environment variables. For more information about the **tso** command and the environment variables associated with it, see *OS/390 UNIX System Services Command Reference*.

Online Help

Two help facilities are available with the shell:

- The **man** command, which you can use to display help information about a shell command. The man page is displayed in your shell session, and you can work in the shell while viewing the help information.
- The TSO/E OHELP command, which displays online reference information about shell commands, TSO/E commands, C functions, callable services, and messages issued by the shell and **dbx**.

  The IBM BookManager READ product is a requirementfor OHELP. The help information is displayed in a BookManager session; while viewing the help information, you cannot work in the shell.

Using the man Command

You can use the **man** command to get help information about a shell command. The **man** syntax is:

man *command_name*

- To scroll the information in a man page, press <Enter>.
- To end the display of a man page, type **q** and press <Enter>.

To search for a particular string in a system that has a list of one-line command descriptions, use the *-k* option:

man *-k string*

For example, to produce a list of all the shell commands for editing, you could type:

```
man -k edit
```

You can use the **man** command to view manual descriptions of TSO/E commands. To do this, you must prefix all commands with **tso**. For example, to view a description of the MOUNT command, you would enter:

```
man tsomount
```

You can also use the **man** command to view manual descriptions of **dbx** subcommands. To do this, you must prefix all subcommands with **dbx**. For example, to view a description of the **dbx alias** subcommand, you would enter:

```
man dbxalias
```

For complete information about the **man** command, see *OS/390 UNIX System Services Command Reference*.

Using the OHELP Command

The TSO/E OHELP command provides a similar capability to the **man** shell command. OHELP displays online reference information about commands, C functions, callable services, and messages issued by the shell and **dbx**. For information on entering TSO/E commands in TSO/E, the shell, and ISPF, see "Entering a TSO/E Command" on page 202.

Your system must have the BookManager READ productinstalled for you to use OHELP.

The OHELP syntax is:

```
OHELP ref_id search_item
```

ref_id A number that specifies the online book to be searched. The default is 1 for *OS/390 UNIX System Services Command Reference*. Each installation can define which number is associated with each book. To see the list of available books and the number associated with each book, type `ohelp`.

search_item

This can specify a:

- Command name
- C function name
- Callable service name
- Message number
- Text string (enclosed in double quotes)

If you omit this operand, OHELP displays the table of contents of the book specified by the `ref_id`.

Example: Getting Help for a Command

For example, if you want information on the **cp** shell command, you would enter:

```
OHELP cp
```

(You do not need to enter the value 1 because 1 is the default.)

The output displayed looks similar to Figure 14 on page 90:

```
                      List All Topics with Matches

Fuzzy matches for: CP
                                              Search matches 1 to 13 of 13
2.33     cp -- Copy a file
A.0      Appendix A.  OS/390 Shell Command Summary
2.21     chcp -- Set or query ASCII/EBCDIC code pages for the terminal
FRONT_1  Permuted Index
2.17     cat -- Concatenate or display a text file
2.89     ln -- Create a link to a file
2.107    mv -- Rename or move a file or directory
2.133    rm -- Remove a directory entry
2.159    touch -- Change the file access and modification times
2.162    trap -- Intercept abnormal conditions and interrupts
2.34     cpio -- Copy in/out file archives
2.42     dd -- Convert and copy a file
2.184    vi -- Use the display-oriented interactive text editor


Command ===> _____________________________________________ SCROLL ===> PAGE
 F1=Help     F4=Text     F5=No Text  F6=Review   F7=Bkwd     F8=Fwd
F10=Explain F12=Cancel
```

Figure 16. Sample Output from the Command OHELP cp

When you look at the output, you can see a boxed display overlaying another display. The boxed display, titled “List All Topics with Matches” lists all references to the **cp** command in the online *OS/390 UNIX System Services Command Reference*.

- `Fuzzy matches for: CP` is the heading for the list of references to **cp** that were found. BookManager converts the shell command name to uppercase.
- `Search Matches 1 to 13 of 13` indicates that this boxed display contains all of the search matches. If there were a very long list of search matches, you would need to scroll to the next screen to get to the end of the list.
- If you press PF4 (PF4=TEXT) while viewing the list, an explanation of the reason for the match is displayed.
- Your cursor is under the first item in the boxed display: `2.33 cp`. This is the **cp** command description from *OS/390 UNIX System Services Command Reference*. The first item in the list is usually the reference information for the language element you specified. Press <Enter>. You can read through the entire command description.
- To redisplay the boxed display of the search results, type `search cp`. Press <Enter>. Alternatively, you can position your cursor under the selection `Search` at the top of the screen. Press <Enter>. On the pulldown menu, select `List all topics with matches` and press <Enter>.
- After you select a match, you can use type `find cp` to move to the next match

If you press the Cancel function key, the boxed display disappears and you see the underlying information: the table of contents for the online *OS/390 UNIX System Services Command Reference*.

To exit the online help, use the Cancel and Exit function keys, as appropriate, from each panel.

Example: Searching Help for All Instances of a Language Element Name

If you want to look at the reference information for all types of language element with the name **chmod**, you enter the command:

```
ohelp * chmod
```

The output displayed would look similar to this:

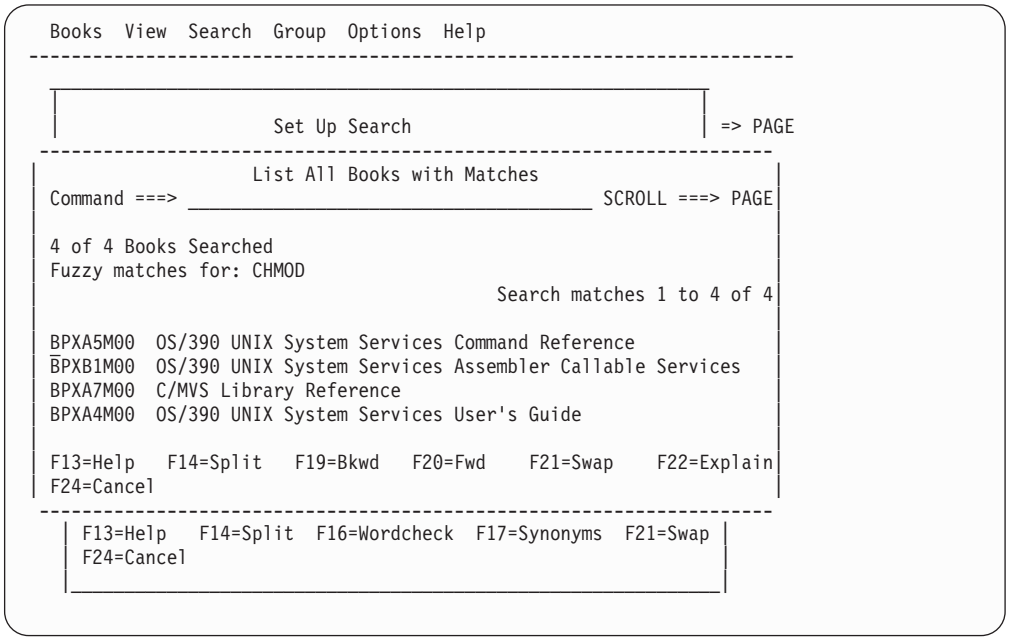


*Figure 17. Sample Output from the Command OHELP * chmod*

When you look at the output, you see a boxed display overlaying another display. The boxed display, titled “List All Books with Matches” lists all the reference books that document a language element named **chmod** command.

- `4 of 4 Books Searched` indicates that four books were searched.
- `Fuzzy matches for: CHMOD` is the heading for the list of references to **chmod** that were found. BookManager converts the shell command name to uppercase.
- `Search matches 1 to 4 of 4` indicates that this boxed display contains all of the search matches.
- Your cursor is under the first book in the list: `BPXA5M00`. If you press <Enter>, you see a boxed display similar to the one shown in Figure 14 on page 90, showing all search matches for **chmod** in the online *OS/390 UNIX System Services Command Reference*. The first item in the list is usually the reference information for the language element you specified. Press <Enter>. You can read through the entire function description.
- To return to the boxed display from the reference information, position your cursor under the selection `Search` at the top of the screen. Press <Enter>. On the pulldown menu, select `List all topics with matches` and press <Enter>.
- The remaining items listed are cross-references to the **chmod** function throughout the online *OS/390 UNIX System Services Command Reference*.

If you press <F12>, the boxed display disappears and you see the “Set Up a Search” panel, which allows you to search for a different name.

To exit the online help, use <F12> and <F3> as appropriate.

Searching for a Text String

To search for a text string, enclose the text in double quotes and specify the `ref_id` for the specific book you want to search. For example, the command

```
ohelp 4 "improper type"
```

will search the book *OS/390 UNIX System Services Messages and Codes* for messages that contain the text *improper type*.

If you are searching for a text string and you use an `*` for `ref_id`, OHELP will search all the books on the shelf and locate every instance of that string.

Shell Messages

Messages issued by the tcsh shell and utilities are prefixed with the letters FSUC. To display online reference information about any shell message, use the OHELP command. The shell messages are documented in *OS/390 UNIX System Services Messages and Codes*.

Chapter 8. Writing OS/390 Shell Scripts

Most people find themselves using some sequences of commands over and over again.

- A programmer may always use the same commands to compile source code, and link the resulting object code.
- A bookkeeper may have to go through the same sequence of shell commands each week to update the books and produce a report.

To simplify such jobs, the shell lets you run a sequence of commands that have been stored in a text file. For example, the programmer could store all the appropriate compiling and linking commands in a file. A file containing commands in this way is called a *shell script*. After such a file is completed and it is made "executable," the programmer can run all the commands in the file by entering the filename on the command line.

Putting commands in a shell script has several advantages over typing the commands individually. Using a shell script:

- Reduces the amount of typing you have to do. You have to type in the shell script only once. Then you can run all the commands in the script by entering the name of the file as a single shell command. A shell script can save you a lot of time and effort if you are working with many files, or if some command lines have several options.
- Reduces the number of errors. If you are typing in ten commands, you have ten chances to make a mistake. With a shell script, however, you can take your time, edit the file carefully, and get it right before you try to run it.
- Makes it easy for other people to do what you do. For example, consider the bookkeeper mentioned earlier. When the bookkeeper goes on vacation, someone else has to fill in. It is much easier for the substitute bookkeeper to type a single command that does everything correctly than to try to type in the full sequence of commands.

For all these reasons, you will probably find that the use of shell scripts makes your work easier and more productive. This chapter can provide only a brief overview, but it should give you an idea of how to write and use shell scripts.

Running a Shell Script

You can run a shell script by typing the name of the file that contains the script. For example, suppose you have a script named **totals.scp** that has three shell commands in it. If you enter:

```
totals.scp
```

the shell runs the three commands.

Before you can run a shell script, you must have read and execute permission to the file. Use the **chmod** and **umask** commands to set the permissions. See the discussion of permissions in Chapter 16 and the command descriptions in *OS/390 UNIX System Services Command Reference*.

For another example, suppose you want to compile a collection of files written in the C programming language. You could use the **c89**, **cc**, or **c++** command. The

c89 command, for example, compiles any file **file.c**, link-edits the object module, and produces an executable file. The shell script:

```
c89 -c file1.c file2.c                      # compile only
c89 -o outfile file1.o file2.o file3.c      # outfile for executable
```

compiles and link-edits the files and produces an executable file, **outfile**. Notice that in a shell script you precede a comment with a #.

If you store this script in an executable file named **compile**, it could be run with the single command **compile**. A new process is created for the script to run in.

To run a shell script in your current environment, without creating a new process, use the **.** (dot) command. You could run the **compile** shell script this way:

```
. compile
```

Should you want to use a shell script that updates a variable in the current environment, run it with the **.** command.

Improving Shell Script Performance: You can improve shell script performance by setting the **_BPX_SPAWN_SCRIPT** environment variable to a value of YES. See "Improving the Performance of Shell Scripts" on page 44 for more information.

Using the Magic Number

When a script file starts with #!, the kernel's spawn and exec services recognize the file name after the #! as the program to be run. For example, HFS file **/u/userid/util1** contains the following in the start of the file:

```
#!  /u/userid/othershell
```

The kernel recognizes the magic number (#!) and runs **/u/userid/othershell**.

Using TSO/E Commands in Shell Scripts

A shell script can include TSO/E commands as well as shell commands, and it can process TSO/E command output. You use the **tso** shell command to run the TSO/E command. For a discussion of the **tso** command, see "Using the tso Command" on page 88.

Using Variables

You can think of shell scripts as *programs* made up of shell commands. To allow more versatile shell scripts, the shell supports many of the features of normal programming languages.

In a conventional programming language, a *variable* is a name that has an associated value. When you want to use the value, you can use the name instead.

Note: A shell script does not inherit any variables from your current shell session. To pass on a variable, you must export it.

Creating a Variable

The shell also lets you create variables. A shell variable name can consist of uppercase or lowercase letters, plus digits and the underscore character _. The

name can have any length, but the first character cannot be a digit. Uppercase letters are distinguished from lowercase ones, so **NAME**, **name**, and **Name** are all *different* names.

To create a shell variable, just enter:

```
name='string'
```

as a command to the shell. No spaces are allowed around the **=**. For example:

```
HOME='/usr/macneil'
```

sets up a variable with the name **HOME** and the value **/usr/macneil**.

After you set a variable, you refer to it by prefixing its name with a dollar sign ($). Any command can use the value of a variable by referring to it this way. For example, if **HOME** is set to **/usr/macneil**:

```
cd $HOME
```

is equivalent to:

```
cd /usr/macneil
```

Similarly:

```
cp $HOME/* /newdir
```

is equivalent to:

```
cp /usr/macneil/* /newdir
```

To change the value of an existing variable, you use a command with the same form as the existing variable. For example:

```
HOME='/usr/benjk'
```

changes the value of **HOME** from **/usr/macneil** to **/usr/benjk**.

If the value on the right-hand side of the **=** sign does not contain spaces, tab characters, or other special characters, you can leave out the single quotes. For example, you can enter:

```
HOME=/usr/benjk
```

Calculating with Variables

Suppose you run the following commands either in a shell script or by typing in one command after another:

```
i=1
j=$i+1
echo $j
```

The output of **echo** is `1+1` because a normal variable assignment assigns a *string* to a variable. Thus `j` gets the string `1+1`.

To *evaluate* an arithmetic expression, you can enter:

```
let "variable=expression"
```

This command line assigns the value of an expression to the given variable. For example:

```
i=1
let "j=$i+1"
echo $j
```

Here `j` is assigned the value of the expression and the **echo** command displays the value `2`.

You can also use **let** to change the value of a variable. If you enter:

```
i=1
let "i=$i+1"
echo $i
```

the **let** command *changes* the value of `i`. The new value of `i` is the old value plus 1.

A **let** command can have any of the standard arithmetic expressions:

-A Negative `A`

A*B `A` times `B`

A/B `A` divided by `B`

A%B Remainder of `A` divided by `B`

A+B `A` plus `B`

A-B `A` minus `B`

The standard mathematical order of operations is used, as shown in the way that operations are grouped:

- All unary minus operations are carried out;
- Then any `*`, `/`, or `%` operations (from left to right in the order they appear);
- Then any additions or subtractions (from left to right in the order they appear).

Many operators use special shell characters, so you usually need to put double quotes around the expression. Thus:

```
let "i=5+2*3"
```

assigns 11 to `i`, since the multiplication is done first. You can use parentheses in the usual way to change the order of operations. For example:

```
let "i=(5+2)*3"
```

assigns 21 to `i`.

Note: **let** does not work with numbers that have fractional parts. It works only with integers.

Exporting Variables

Up to this point, we have talked about defining shell variables and then using them in later command lines. You can also define a shell variable and then call a shell script that makes use of that variable. But you have to do a certain amount of preparation first.

A shell script is run like a separate shell session. By default, it does not share any variables with your current shell session. If you define a variable **VAR** in the current session, it is *local* to the current session; any shell script that you call will not know about **VAR**.

To deal with this situation, you can export the command; enter:

```
export VAR
```

The **export** command says that you want the variable **VAR** passed on to all the commands and shell scripts that you execute in this session. After you do this, **VAR** becomes *global* and the variable is known to all the commands and shell scripts that you use.

As an example, suppose you enter the commands:

```
MYNAME="Robin Hood"
export MYNAME
```

Now all your commands can use the **MYNAME** variable to obtain the associated name. You may, for example, have shell scripts that write form letters that contain your name, `Robin Hood`, obtained from the **MYNAME** variable.

Note: You could use single or double quotes to enclose the variable value. See "Quoting Variable Values" on page 40 for more information.

When a script begins running, it automatically inherits all the variables currently being exported. However, if the script changes the value of one of those variables, that change is *not* reflected to the calling shell—unless you run the script with the dot (•) utility.

By default, any variables created within a shell script are *local* to that script. This means that when another program is run, those variables do not apply in its environment. However, the script can use the **export** command to turn local variables into global ones. Inside a shell script:

```
export name
```

indicates that the variable with the given *name* should be exported. When other programs are run from that script, they inherit the value of all exported variables. However, when the script ends, all its exported variables are lost to the calling shell.

Some variables are automatically marked for export by the software that creates them. For example, if you invoke the shell, the initialization procedure automatically marks the **HOME** variables for export so that other commands and shell scripts can use it. In Chapter 4, you saw that in a typical **.profile** file for an individual user, the **PATH** variable is exported. Exporting **PATH** ensures that search rules and changes to search rules are automatically shared by all shell sessions and scripts.

You must export other variables explicitly, using the **export** command.

Associating Attributes with Variables

The **typeset** command lets you associate attributes with shell variables. This process is analogous to declaring the type of a variable in a conventional programming language. For example:

```
typeset –i8 y
```

says that `y` is an octal integer. In this way, you can make sure that arithmetic with `y` is always performed in base 8 rather than the usual base 10.

Other attributes may specify how the variable's value is displayed when the variable is expanded. Attributes of this kind are:

–L*n* The value should always be displayed with *n* characters, left-justified within that space.

–R*n* The value should always be displayed with *n* characters, right-justified within that space.

–RZ*n* The value should always be displayed with *n* characters, right-justified and with enough leading zeros to fill out the rest of the space.

–Z*n* The same as **-RZ***n*.

–LZ*n* The value should always be displayed with *n* characters, left-justified and with leading zeros stripped off.

All of these options may lead to truncation of a value that is longer than the specified length.

You can use the **–u** attribute of **typeset** for variables with string values. Then whenever such a variable is assigned a new value, all lowercase letters in the value are automatically converted to uppercase. Similarly, the **–l** attribute specifies that whenever a variable is assigned a new value, all uppercase letters in the value are automatically converted to lowercase.

The read-only attribute **–r** is useful when a variable is marked for export. The command:

```
typeset –r name
```

says that the variable *name* cannot be changed from its present value. Then subsequent commands cannot change this value. You can also use the format:

```
typeset –r name=value
```

which sets the variable to the given value and then marks it read-only so that the value cannot be changed.

Displaying Currently Defined Variables

The command **typeset** without any arguments displays the currently defined variables and their attributes. The variation:

```
typeset -x
```

displays all the variables currently defined for export.

Using Positional Parameters — the $N Construct

The sample shell script discussed earlier in this chapter compiled and link-edited a program stored in a collection of source modules. This section discusses a shell script that can compile and link-edit a C program stored in any file.

To create such a script, you need to be familiar with the idea of *positional parameters*. When the shell encounters a `$N` construct formed by a `$` followed by a single digit, it replaces the construct with a value taken from the command line that started the shell script.

- `$1` refers to the first string after the name of the script file on the command line
- `$2` refers to the second string, and so on.

As a simple example, consider a shell script named **echoit** consisting only of the command:

```
echo $1
```

Suppose we run the command:

```
echoit hello
```

The shell reads the shell script from **echoit** and tries to run the command it contains. When the shell sees the `$1` construct in the **echo** command, it goes back to the command line and obtains the first string following the name of the shell script on the command line. The shell replaces the `$1` with this string, so the **echo** command becomes:

```
echo hello
```

The shell then runs this command.

A construct like `$1` is called a *positional parameter*. Parameters in a shell script are replaced with strings from the command line when the script is run. The strings on the command line are called *positional parameter values* or *command-line arguments*.

If you enter:

```
echoit Hello there
```

the string `Hello` is considered parameter value `$1` and `there` is `$2`. Of course, the shell script is only:

```
echo $1
```

so the **echo** command displays only the `Hello`.

Positional parameters that include a blank can be enclosed in quotes (single or double). For example:

```
echoit "Hello there"
```

echoes the two words instead of just one, because the two words are handled as one parameter.

Returning to a compile and link example, a programmer could write a more general shell script as:

```
c89 -c $1.c
c89 -o $1 $1.o
```

If this shell script were named **clink**, the command:

```
clink prog
```

would compile and link **prog.c**, producing an executable file named **prog** in the working directory. In the same way, the command:

```
clink dir/prog2
```

would compile and link **dir/prog2.c**. The shell script compiles and links a C program stored in a single file.

As another example of a shell script containing a positional parameter, suppose that the file **lookup** contains:

```
grep $1 address
```

(where **address** is a file containing names, addresses, and other useful information). The command:

```
lookup Smith
```

displays address information on anyone in the file named Smith.

Using Quotes to Enclose a Construct in a Shell Script

A `$N` construct in a shell script can be enclosed in double or single quotes.

- When *double* quotes are used, the parameter is replaced by the appropriate value from the command line. For example, suppose the file **search** contains:

  ```
  grep "$1" *
  ```

  If you enter the command:

  ```
  search 'two words'
  ```

  the parameter value `'two words'` replaces the construct `$1` in the **grep** command:

  ```
  grep "two words" *
  ```

  If the **grep** command does not contain the double quotes, the parameter replacement would result in:

  ```
  grep two words *
  ```

  which has an entirely different meaning.
- When you use *single* quotes to enclose a `$N` construct in a shell script, the `$N` is *not* replaced by the corresponding parameter value. For example, if the file **search** contains:

  ```
  grep '$1' *
  ```

  grep searches for the string `$1`. The `$1` is not replaced by a value from the command line. In general, single quotes are “stronger” than double quotes. Less is more!

Using Parameter and Variable Expansion

As we just discussed, a `$` followed by a number stands for a positional parameter passed to the script or function. A positional parameter is represented with either a single digit (except 0) or two or more digits in curly braces; for example, 7 and {15} are both valid representations of positional parameters. For example, if the command:

```
echo $1
```

appeared in a shell script, it would **echo** the first positional parameter.

Similarly, a `$` followed by the name of a shell variable (such as **$HOME**) stands for the value of the variable.

These constructs are called *parameter expansions*. In this sense, the term *parameter* can mean either a positional parameter or a shell variable.

The OS/390 shell also supports more complicated forms of parameter expansions, letting you obtain only part of a parameter value or a modified form of the value.

Parameter Expansion	Usage

${parameter:-value}	You can use ${*parameter*:-*value*} in any input to the shell. If *parameter* currently has a value and the value is not null (for example, a string without characters), the foregoing construct stands for the parameter's value; if the value of the parameter is null, the construct is replaced with the *value* shown in the brace brackets. For example, a shell script might contain:
	`SHELL=${SHELL:-/bin/sh}`
	
	If the **SHELL** variable currently has a value, this simply assigns **SHELL** its own current value. However, if the value of **SHELL** is null, the above assignment gives it the value of **/bin/sh**. The value after :– can be thought of as a *backup* value in case the parameter itself does not have a value. As another example, consider:
	`cp $1 ${2:-$HOME}`
	
	(This might occur in a shell script.) If both positional parameters are present and have a nonnull value, the copy command is just:
	`cp $1 $2`
	
	However, if you call the shell script without specifying a second positional parameter, it uses the backup value of **$HOME**. The result is equivalent to:
	`cp $1 $HOME`

${parameter:=value}	The expansion form ${*parameter*:=*value*} is similar to the previous form; the difference is that if the given *parameter* does not currently have a value, the given *value* is assigned to *parameter*, and then the new value of parameter is used. Thus the := form actually assigns a value if the *parameter* does not already have one. In this case, *parameter* must be a variable; it cannot be a positional parameter.

${parameter:?message}	The expansion ${*parameter*:?*message*} is related to the previous two forms. If the value of the given *parameter* is null, the given *message* is displayed. If the construct is being used inside a shell script, the script ends with an error status. For example, you might have:
	`cp $1 ${2:?"Must specify a directory name"}`
	
	In this case, the message following the ? is displayed if there is no second positional parameter. If you omit the *message*, the shell prints a standard message. For example, you could just enter:
	`cp $1 ${2:?}`
	
	to get the standard error message.

Parameter Expansion	Usage

${parameter:+replacement}	The construct ${*parameter*:+*replacement*} might be thought of as the opposite of the preceding expansions. If *parameter* has not been assigned a value, or has a null value, this construct is just the null string. If *parameter does* have a value, the value is ignored and the *replacement* value is used in its place. Thus, if a shell script contains:
	`echo ${1:+"There was a parameter"}`
	
	the **echo** command displays:
	`There was a parameter`
	
	if the script was invoked with a parameter. If no parameter was specified, the **echo** command has nothing to echo.

${parameter#pattern}	The construct ${*parameter#pattern*} is evaluated by expanding the value of *parameter* and then deleting the *smallest leftmost* part of the expansion that matches the given *pattern* of pathname wildcard characters. For example, suppose that the variable *NAME* stands for a filename. You might use:
	`${NAME#*/}`
	
	to remove the highest-level directory from the pathname. If:
	`NAME="user/dir/subdir/file.c"`
	
	then:
	`${NAME#*/}`
	
	expands to:
	`dir/subdir/file.c`

${parameter##pattern}	The construct ${*parameter##pattern*} removes the *largest leftmost* part that matches the pattern. For example, if:
	`NAME="user/dir/subdir/file.c"`
	
	then:
	`${NAME##*/}`
	
	yields:
	`file.c`
	
	The wildcard character * stands for any sequence of characters. In this situation, it stands for everything up to the final slash.

${parameter%pattern}	The construct ${*parameter%pattern*} removes the *smallest rightmost* part of the parameter expansion that matches *pattern.* Thus if:
	`NAME="user/dir/subdir/file.c"`
	
	then:
	`${NAME%.?}`
	
	stands for:
	`user/dir/subdir/file`

Parameter Expansion	Usage

${parameter%%pattern}	Similarly, ${*parameter%%pattern*} stands for the expansion of *parameter* without the *longest rightmost* string that matches *pattern*. Using the above example of *NAME*,
	`${NAME%%/*}`
	
	stands for:
	`user`

Using Special Parameters in Commands and Shell Scripts

The OS/390 shell has a variety of special parameters that may be used in command lines and shell scripts.

Parameter	Expands to

$@	The complete list of positional parameters, each separated by a single space. If **$@** is quoted, the separate arguments are each quoted; for example:
	`echo "$@"`
	
	is equivalent to:
	`"$1" "$2" "$3"`
	
	If the positional parameters are all filenames:
	`cp $@ dir`
	
	copies all the files to the given directory **dir**.

$*	The complete list of positional parameters. If **$*** is quoted, the result is concatenated into a single argument, with parameters separated by the first character of the value of the shell variable **IFS**. For example, if the first character of **IFS** is a comma, then:
	`echo "$*"`
	
	displays the parameters with separating commas:
	`"$1,$2,$3"`

$#	The number of positional parameters passed to this shell script. This number can be changed by several shell commands (for example, **set** or **shift**); see *OS/390 UNIX System Services Command Reference*.
$?	The exit status value returned by the most recently run command. The command **echo $?** prints out the status from the most recently run operation or command.
$–	The set of options that have been specified for this shell session. This includes options that were specified on the command line that started the shell, plus other options that have been set with the **set** command.

Using Control Structures

The shell provides facilities similar to those found in programming languages. It offers these *control structures,* which are related to programming control structures:

- The **if** conditional
- The **while** loop
- The **for** loop

Using test to Test Conditions

Before discussing the various control structures, it is useful to talk about ways to test for various conditions.

The **test** command tests to see if something is true. Here are some ways it can be used:

Examine the nature of a file	
test -d *pathname*	Is *pathname* a directory?
test -f *pathname*	Is *pathname* a file?
test -r *pathname*	Is *pathname* readable?
test -w *pathname*	Is *pathname* writable?
Compare the age of two files	
test *file1* **-ot** *file2*	Is *file1* older than *file2*?
test *file1* **-nt** *file2*	Is *file1* newer than *file2*?
Compare the values of numbers *A* **and** *B*	
test *A* **-eq** *B*	Is *A* equal to *B*?
test *A* **-ne** *B*	Is *A* not equal to *B*?
test *A* **-gt** *B*	Is *A* greater than *B*?
test *A* **-lt** *B*	Is *A* less than *B*?
test *A* **-ge** *B*	Is *A* greater than or equal to *B*?
test *A* **-le** *B*	Is *A* less than or equal to *B*?
Compare two strings *str1* **and** *str2*	
test *str1* **=** *str2*	Is *str1* equal to *str2*?
test *str1* **!=** *str2*	Is *str1* not equal to *str2*?
Test whether strings are empty	
test -z *string*	Is *string* empty?
test -n *string*	Is *string* not empty?

Any of these tests will also work if you put square brackets ([ ]) around the condition instead of using the **test** command. For example, `test 1 -eq 1` is the equivalent of `[ 1 -eq 1 ]`.

With Release 8 and later, the double square bracket `[[test_expr]]` syntax is supported. The double square bracket ([[ ]]) also supports additional tests over the **test** command, and there are some subtle differences between the tests (for example, string equal vs. pattern matching).

The result of **test** is either true or false. (To be precise, **test** returns a status of `0` if the test turns out to be true and a status of `1` if the test turns out to be false.)

You can use **–n** to check if a variable has been defined. For example:

```
test -n "$HOME"
```

is true if **HOME** exists, and false if you have not created a **HOME** variable.

You can use **!** to indicate logical negation;

```
test ! expression
```

returns false if *expression* is true, and returns true if *expression* is false. For example:

```
test ! -d pathname
```

is true if *pathname* is not a directory, and false otherwise.

The if Conditional

An **if** conditional runs a sequence of commands if a particular condition is met. It has the form:

```
if condition
then commands
fi
```

The end of the commands is indicated by **fi** (which is "if" backward). For example, you could have:

```
if test -d $1
then ls $1
fi
```

This tests to see if the string associated with the first positional parameter, `$1`, is the name of a directory. If so, it runs an **ls** command to display the contents of the directory.

Any number of commands may come between the **then** and the **fi** that ends the control structure. For example, you might have written:

```
if
    test -d $1
then
    echo "$1 is a directory"
    ls $1
fi
```

This example also shows that the commands do not have to begin on the same line as **then**, and the condition being tested does not have to begin on the same line as **if**. The condition and the commands are indented to make them stand out more clearly. This is a good way to make your shell scripts easier to read.

Another form of the **if** conditional is:

```
if condition
then commands
else commands
fi
```

If the condition is true, the commands after the **then** are run; otherwise, the commands after the **else** are run. For example, suppose you know that the string associated with the variable *pathname* is the name of either a directory or a file. Then you could write:

```
if
    test -d $pathname
then
    echo "$pathname is a directory"
    ls $pathname
else
    echo "$pathname is a file"
    cat  $pathname
fi
```

If the value of *pathname* is the name of a file, this shell script uses **echo** to display an appropriate message, and then uses **cat** to display the contents of the file.

The final form of the **if** control structure is:

```
if condition1
then commands1
elif condition2
then commands2
elif condition3
then commands3
      ...
else commands
fi
```

elif is short for "else if." In this example, if *condition1* is true, *commands1* are run; otherwise, the shell goes on to check *condition2*. If that is true, *commands2* are run; otherwise, the shell goes on to check *condition3* and so on. If none of the test conditions are true, the *commands* after the **else** are run. Here is an example of how this can be used:

```
if test ! "$1"
then
    echo "no positional parameters"
elif test -d $1
then
    echo "$1 is a directory"
    ls $1
elif test -f $1
then
    echo "$1 is a file"
    cat $1
else
    echo "$1 is just a string"
fi
```

The test after the **if** determines if the value of the first positional parameter, `$1`, is an empty string. If so, there are no positional parameters, so the shell script uses **echo** to display an appropriate message; otherwise, the script checks to see if the parameter is a directory name; if so, the contents of the directory are listed with **ls** (after an appropriate message). If that does not work, the script checks to see if the parameter is a filename; if so, the contents of the file are listed with **cat** (after an appropriate message). Finally, if none of the previous tests work, the parameter is assumed to be an arbitrary string, and the script displays a message to this effect.

You could put that script into a file named **listit** and run commands of the form:

```
listit name
```

to list the contents of *name* in a useful form.

The while Loop

The **while** loop repeats one or more commands while a particular condition is true. The loop has the form:

```
while condition
do commands
done
```

The shell first tests to see if *condition* is true. If it is, the shell runs the *commands*. The shell then goes back to check the *condition*. If it is still true, the shell runs the *commands* again, and so on, until the *condition* is found to be false.

As an example of how this can be used, suppose you want to run a program named **prog** 100 times to get an idea of the program's average running speed. The following shell script does the job:

```
i=100
date
while test $i -gt 0
do
    prog
    let i=$i-1
done
date
```

The script begins by setting a variable *i* to 100. It then uses the **date** command to get the current date and time.

Next the script runs a **while** loop. The **test** condition says that the loop should keep on going as long as the value of *i* is greater than zero. The commands of the loop run **prog** and then subtract 1 from the *i* variable. In this way, *i* goes down by 1 each time through the loop, until it is no longer greater than 0. At this point, the loop stops and the final instruction of the script prints out the date and time at the end of the loop. The difference between the starting time and the ending time should give some idea of how long it took to run the program 100 times.

(Of course, the shell itself takes some time to perform the **test** and to do the calculations with *i*. If **prog** takes a long time to run, the time spent by the shell is relatively unimportant; if **prog** is a quick program, the extra time that the shell takes may be large enough to make the timing incorrect.)

You can rewrite this shell script to make it a little more efficient:

```
i=100
date
while let "(i=$i-1) >= 0"
do
    prog
done
date
```

In this example, the **let** command is the condition of the **while** loop. It gives *i* a new value and then compares this value to zero. The advantage of this way of writing the program is that it does not have to call **test** to make the comparison; this speeds up the loop and makes the time more accurate.

The for Loop

The final control structure to be examined is the **for** loop. It has the form:

```
for name in list
do commands
done
```

The parameter *name* should be a variable name; if this variable doesn't exist, it is created. The parameter *list* is a list of strings separated by spaces. The shell begins by assigning the first string in *list* to the variable *name*. It then runs the *commands* once. Then the shell assigns the next string in *list* to *name*, and repeats the *commands*. The shell runs the *commands* once for each string in *list*.

As a simple example of a shell script that uses **for**, consider:

```
for file in *.c
do
    c89 $file
done
```

When the shell looks at the **for** line, it expands the expression `*.c` to produce a *list* containing the names of all files (in the working directory) that have the suffix **.c**. The variable *file* is assigned each of the names in this list, in turn. The result of the **for** loop is to use the **c89** command to compile all **.c** files in the working directory. You could also write:

```
for file in *.c
do
    echo $file
    c89 $file
done
```

so that the shell script displayed each filename before compiling it. This would let you keep track of what the script was doing.

As you can see, the **for** loop is a powerful control structure. The *list* can also be created with command substitution, as in:

```
for file in $(find . -name "*.c" -print)
do
    echo $file
    c89 $file
done
```

Here the **find** command finds all **.c** files in the working directory, and then compiles these files. This is similar to the previous shell script, but also looks at subdirectories of the working directory.

Combining Control Structures

You can combine control structures by nesting (that is, putting one inside another). For example:

```
for file in $(find . -name "*.c" -print)
do
    if test $file -ot $1
    then
        echo $file
        c89 -c $file
    fi
done
```

This shell script takes one positional parameter, giving the name of a file. The script looks in the working directory and finds the names of all **.c** files. The **if** control structure inside the **for** loop tests each file to see if it is older than the file named on the command line. If the **.c** file is older, **echo** displays the name, and the file is compiled. You can think of this as making a set of files up to date with the filename specified on the command line.

Using Functions

A *shell function* is similar to a function in C: It is a sequence of commands that do a single job. Typically, a function is used for an operation that you tend to do frequently in a shell script. Before you can call a function in a shell script, you must define it in the script. After the function is defined, you can call it as many times as you want in the script.

As an example, consider the following piece of a shell script, showing the function definition and how the function is called in the shell script:

```
function td
{
    if  test -d "$1"               # test if first argument is directory
    then
        curdir=$(pwd)              # set curdir to working directory
        cd $1                      # change to specified directory
        $2                         # run specified command
        cd $curdir                 # change back to working directory
        return 0                   # return 0 if successful
    else
        echo $1 "is not a directory"
        return 1                   # return 1 if not successful
    fi
}
td /u/turbo/src.c ls               # invoking the function
```

The purpose of **td** is to go to a specified directory, run a single command, and then return to the directory from which the function was called.

To run a function, specify the function's name followed by whatever arguments it expects. To run the function **td**, specify the function name followed by a directory name and a command name, as shown in the last line of the foregoing example.

As you see in the **td** example, a function can also return a value. If the statement:

`return` *`expression`*

appears inside a function, the function ends and the value of *expression* is returned as the status, or *result,* of the function. In general, the returned value:

- `0` means that the function has succeeded in its task.
- `1` means that the function has failed.

Anytime you need to repeatedly perform the same sequence of commands in a shell script, consider defining a function to do the sequence of commands. This lets you organize a large script into smaller blocks of subroutines.

In order to make a shell function available as a shell command, the function definition must be processed by the shell that will execute the command. Typically, the user sets up a shell script (e.g. $HOME/.setup) that contains all of the function definitions, and sets the ENV variable to the pathname of that shell script. As the number of functions in this script grows, the time to process the function definitions causes shell initialization time to increase.

Autoloading Functions

Autoloading improves the performance of shell initialization by delaying function definition processing until the first use. Functions that are not used by a particular user are never read by the shell, thus avoiding the processing of unused functions. The **FPATH** variable allows flexibility in accessing directories with system-wide, group, or personal function definitions.

FPATH is defined with the same format as the **PATH** variable. **FPATH** is a list of directories separated by colons. These directories contain only function definitions and should not contain the current working directory.

To use autoloading, place frequently used and shared functions in a directory pointed to by the **FPATH** variable and specify the function name on an **autoload** or **typeset —f** command in the user's ENV setup script.

The **autoload** command identifies functions that are not yet defined. The first time that an **autoload** function is called within the shell, the shell searches **FPATH** directories for a file with the same name as the function definition. If a matching file with the same name as the function is found, it is processed and stored in the shell's memory for subsequent execution. The matching file contains the function definition for the **autoload** function. Other function definitions may be found in this matching file, and if so, they will be defined to the shell when the file is processed. For information about how to set up the **FPATH** search path, see "Customizing the FPATH Search path: the FPATH Variable" on page 44.

Chapter 9. Writing tcsh Shell Scripts

Most people find themselves using some sequences of commands over and over again.

- A programmer may always use the same commands to compile source code, and link the resulting object code.
- A bookkeeper may have to go through the same sequence of shell commands each week to update the books and produce a report.

To simplify such jobs, the shell lets you run a sequence of commands that have been stored in a text file. For example, the programmer could store all the appropriate compiling and linking commands in a file. A file containing commands in this way is called a *shell script*. After such a file is completed and it is made "executable," the programmer can run all the commands in the file by entering the filename on the command line.

Putting commands in a shell script has several advantages over typing the commands individually. Using a shell script:

- Reduces the amount of typing you have to do. You have to type in the shell script only once. Then you can run all the commands in the script by entering the name of the file as a single shell command. A shell script can save you a lot of time and effort if you are working with many files, or if some command lines have several options.
- Reduces the number of errors. If you are typing in ten commands, you have ten chances to make a mistake. With a shell script, however, you can take your time, edit the file carefully, and get it right before you try to run it.
- Makes it easy for other people to do what you do. For example, consider the bookkeeper mentioned earlier. When the bookkeeper goes on vacation, someone else has to fill in. It is much easier for the substitute bookkeeper to type a single command that does everything correctly than to try to type in the full sequence of commands.

For all these reasons, you will probably find that the use of shell scripts makes your work easier and more productive. This chapter can provide only a brief overview, but it should give you an idea of how to write and use shell scripts.

Running a Shell Script

You can run a shell script by typing the name of the file that contains the script. For example, suppose you have a script named **totals.scp** that has three shell commands in it. If you enter:

```
totals.scp
```

the shell runs the three commands.

Before you can run a shell script, you must have read and execute permission to the file. Use the **chmod** and **umask** commands to set the permissions. See the discussion of permissions in Chapter 16 and the command descriptions in *OS/390 UNIX System Services Command Reference*.

For another example, suppose you want to compile a collection of files written in the C programming language. You could use the **c89**, **cc**, or **c++** command. The

c89 command, for example, compiles any file **file.c**, link-edits the object module, and produces an executable file. The shell script:

```
c89 -c file1.c file2.c                      # compile only
c89 -o outfile file1.o file2.o file3.c      # outfile for executable
```

compiles and link-edits the files and produces an executable file, **outfile**. Notice that in a shell script you precede a comment with a #.

If you store this script in an executable file named **compile**, it could be run with the single command **compile**. A new process is created for the script to run in.

To run a shell script in your current environment, without creating a new process, use the **source** command. You could run the **calculate** shell script this way:

```
source calculate
```

Should you want to use a shell script that updates a variable in the current environment, run it with the **source** command.

Improving Shell Script Performance: When using the tcsh shell, the **_BPX_SPAWN_SCRIPT** environment variable should be set to NO. This variable is only intended for use with the OS/390 shell. If this variable is inherited from an OS/390 shell session, put

```
#!/bin/tcsh
```

as the first line in your tcsh shell scripts to avoid any errors. If tcsh is your login shell, you should unset **_BPX_SPAWN_SCRIPT**, since it is only used for increasing performance of OS/390 shell scripts.

Using the Magic Number

All tcsh scripts must have # as the first character of the script. When a script file starts with #!, the kernel's spawn and exec services recognize the file name after the #! as the program to be run. It is recommended that the first line of all tcsh scripts look like:

```
#!/bin/tcsh
```

with /bin/tcsh being the location of tcsh on the OS/390 UNIX system. The kernel recognizes the magic value (#!) and runs **/bin/tcsh**.

Using TSO/E Commands in Shell Scripts

A shell script can include TSO/E commands as well as shell commands, and it can process TSO/E command output. You use the **tso** shell command to run the TSO/E command. For a discussion of the **tso** command, see "Using the tso Command" on page 88.

Using Variables

You can think of shell scripts as *programs* made up of shell commands. To allow more versatile shell scripts, the shell supports many of the features of normal programming languages.

In a conventional programming language, a *variable* is a name that has an associated value. When you want to use the value, you can use the name instead.

Creating a Shell Variable

The shell also lets you create variables. A shell variable name can consist of uppercase or lowercase letters, plus digits and the underscore character _. The name can have any length, but the first character cannot be a digit. Uppercase letters are distinguished from lowercase ones, so **NAME**, **name**, and **Name** are all *different* names.

To create a shell variable, just enter:

```
set name='string'
```

as a command to the shell. For example:

```
set home='/usr/adams'
```

sets up a variable with the name **home** and the value **/usr/adams**.

After you set a variable, you refer to it by prefixing its name with a dollar sign ($). Any command can use the value of a variable by referring to it this way. For example, if **home** is set to **/usr/adams**:

```
cd $home
```

is equivalent to:

```
cd /usr/adams
```

Similarly:

```
cp $home/* /newdir
```

is equivalent to:

```
cp /usr/adams/* /newdir
```

To change the value of an existing variable, you use a command with the same form as the existing variable. For example:

```
set home='/usr/benjk'
```

changes the value of **home** from **/usr/adams** to **/usr/benjk**.

If the value on the right-hand side of the **=** sign does not contain spaces, tab characters, or other special characters, you can leave out the single quotes. For example, you can enter:

```
home=/usr/benjk
```

Calculating with Variables

Suppose you run the following commands either in a shell script or by typing in one command after another:

```
set i=1
set j=$i+1
echo $j
```

The output of **echo** is `1+1` because a normal variable assignment assigns a *string* to a variable. Thus `j` gets the string `1+1`.

To *evaluate* an arithmetic expression, you can enter:

```
@ variable=expression
```

This command line assigns the value of an expression to the given variable. For example:

```
i=1
@ j=$i + 1
echo $j
```

Here `j` is assigned the value of the expression and the **echo** command displays the value `2`.

You can also use **@** to change the value of a variable. If you enter:

```
i=1
@ i=$i + 1
echo $i
```

the **@** command *changes* the value of `i`. The new value of `i` is the old value plus `1`.

An **@** command can have any of the standard arithmetic expressions:

-A Negative `A`

A * B `A` times `B`

A / B `A` divided by `B`

A % B Remainder of `A` divided by `B`

A + B `A` plus `B`

A - B `A` minus `B`

The standard mathematical order of operations is used, as shown in the way that operations are grouped:

- All unary minus operations are carried out;
- Then any `*`, `/`, or `%` operations (from left to right in the order they appear);
- Then any additions or subtractions (from left to right in the order they appear).

Many operators use special shell characters, so you usually need to put double quotes around the expression. Thus:

```
@ i =  5 + 2 * 3
```

assigns 11 to `i`, since the multiplication is done first. You can use parentheses in the usual way to change the order of operations. For example:

```
@ i = ((5 + 2) * 3 )
```

assigns 21 to `i`.

Note: **@** does not work with numbers that have fractional parts. It works only with integers.

Setting Environment Variables

Up to this point, we have talked about defining shell variables and then using them in later command lines. You can also define a shell variable and then call a shell script that makes use of that variable. But you have to do a certain amount of preparation first.

A shell script is run as a child process to the parent shell. By default, the child process does not share any variables with the parent. If you define a variable **var** in the parent shell, it is *local* to the current session; any shell script, or child process, that you call will not inherit **var**.

To deal with this situation, you can enter the following:

```
setenv var [value]
```

The **setenv** command says that you want the variable **var** passed on to all the child processes that you execute in this session. After you do this, **var** becomes inherited and the variable is known to all the commands and shell scripts that you use.

As an example, suppose you enter the commands:

```
setenv myname "Friar Tuck"
```

Now all your child processes can use the **myname** variable to obtain the associated name. You may, for example, have shell scripts that write form letters that contain your name, `Friar Tuck`, obtained from the **myname** variable.

Note: You could use single or double quotes to enclose the variable value. See "Quoting Variable Values" on page 54 for more information.

When a script or child process begins running, it automatically inherits all the environment variables passed on to it. However, if the script changes the value of one of those variables, that change is *not* passed back to the parent process —unless you run the script with the **source** utility.

By default, any variables created within a shell script are *local* to that script. This means that when another program is run, those variables do not apply in its environment. However, the script can use the **setenv** command to turn shell variables into global environment ones. Inside a shell script:

```
setenv name [value]
```

indicates that the variable with the given *name* should be defined as an environment variable. When other programs are run from that script, they inherit the value of all environment variables. However, when the script ends, all its environment variables are lost to the calling shell.

Some variables are automatically inherited by the software that creates them. For example, if you invoke the shell, the initialization procedure automatically marks the **home** variables for environment variables so that other commands and shell scripts can use it. In Chapter 5, you saw that in a typical **.tcshrc** file for an individual user, the **PATH** variable is an environmental variable. Making the **PATH** variable an environmental variable ensures that search rules and changes to search rules are automatically shared by all shell sessions and scripts.

Using Positional Parameters — the $N Construct

The sample shell script discussed earlier in this chapter compiled and link-edited a program stored in a collection of source modules. This section discusses a shell script that can compile and link-edit a C program stored in any file.

To create such a script, you need to be familiar with the idea of *positional parameters*. When the shell encounters a `$N` construct formed by a `$` followed by a single digit, it replaces the construct with a value taken from the command line that started the shell script.

- `$1` refers to the first string after the name of the script file on the command line
- `$2` refers to the second string, and so on.

As a simple example, consider a shell script named **echoit** consisting only of the command:

```
#
echo $1
```

Suppose we run the command:

```
echoit hello
```

The shell reads the shell script from **echoit** and tries to run the command it contains. When the shell sees the `$1` construct in the **echo** command, it goes back to the command line and obtains the first string following the name of the shell script on the command line. The shell replaces the `$1` with this string, so the **echo** command becomes:

```
echo hello
```

The shell then runs this command.

A construct like `$1` is called a *positional parameter*. Parameters in a shell script are replaced with strings from the command line when the script is run. The strings on the command line are called *positional parameter values* or *command-line arguments*.

If you enter:

```
echoit Hello there
```

the string `Hello` is considered parameter value `$1` and `there` is `$2`. Of course, the shell script is only:

```
echo $1
```

so the **echo** command displays only the `Hello`.

Positional parameters that include a blank can be enclosed in quotes (single or double). For example:

```
echoit "Hello there"
```

echoes the two words instead of just one, because the two words are handled as one parameter.

Returning to a compile and link example, a programmer could write a more general shell script as:

```
c89 -c $1.c
c89 -o $1 $1.o
```

If this shell script were named **clink**, the command:

```
clink prog
```

would compile and link **prog.c**, producing an executable file named **prog** in the working directory. In the same way, the command:

```
clink dir/prog2
```

would compile and link **dir/prog2.c**. The shell script compiles and links a C program stored in a single file.

As another example of a shell script containing a positional parameter, suppose that the file **lookup** contains:

```
grep $1 address
```

(where **address** is a file containing names, addresses, and other useful information). The command:

```
lookup Smith
```

displays address information on anyone in the file named Smith.

Using Quotes to Enclose a Construct in a Shell Script

A `$N` construct in a shell script can be enclosed in double or single quotes.

- When *double* quotes are used, the parameter is replaced by the appropriate value from the command line. For example, suppose the file **search** contains:

  ```
  grep "$1" *
  ```

  If you enter the command:

  ```
  search 'two words'
  ```

  the parameter value `'two words'` replaces the construct `$1` in the **grep** command:

  ```
  grep "two words" *
  ```

  If the **grep** command does not contain the double quotes, the parameter replacement would result in:

  ```
  grep two words *
  ```

  which has an entirely different meaning.
- When you use *single* quotes to enclose a `$N` construct in a shell script, the `$N` is *not* replaced by the corresponding parameter value. For example, if the file **search** contains:

  ```
  grep '$1' *
  ```

  grep searches for the string `$1`. The `$1` is not replaced by a value from the command line. In general, single quotes are “stronger” than double quotes. Less is more!

Using Parameter and Variable Expansion

As we just discussed, a $ followed by a number stands for a positional parameter passed to the script or function. A positional parameter is represented with either a single digit (except 0) or two or more digits in curly braces; for example, 7 and {15} are both valid representations of positional parameters. For example, if the command:

```
echo $1
```

appeared in a shell script, it would **echo** the first positional parameter.

Similarly, a $ followed by the name of a shell variable (such as **$HOME**) stands for the value of the variable.

These constructs are called *parameter expansions*. In this sense, the term *parameter* can mean either a positional parameter or a shell variable.

The tcsh shell also supports more complicated forms of parameter expansions, letting you obtain only part of a parameter value or a modified form of the value.

Modifier	Description
r	Root of value
e	Extension of value
h	Head of value
t	Tail of value

For example, to extract only part of a filename, you can add one of the above modifiers as follows:

Filename	r	e	h	t
/usr/bin/vi.txt	/usr/bin/vi	txt	/usr/bin	vi.txt
/u/bobby/mail	/u/bobby/mail	empty	/u/bobby	mail
storybook.pdf	storybook	pdf	empty	storybook.pdf
INSTALL	INSTALL	empty	empty	INSTALL

Using Special Parameters in Commands and Shell Scripts

The tcsh shell has a variety of special parameters that may be used in command lines and shell scripts. These parameters are listed in *OS/390 UNIX System Services Command Reference* under **tcsh** in the ″Variable Substitution″ section.

Using Control Structures

The shell provides facilities similar to those found in programming languages. It offers these *control structures,* which are related to programming control structures:

- The **if** conditional
- The **while** loop
- The **for** loop

The if Conditional

An **if** conditional runs a sequence of commands if a particular condition is met. It has the form:

```
if (expr) command
```

The end of the commands is indicated by **endif**. For example, you could have:

```
if ( -d $1 ) then
ls $1
endif
```

This tests to see if the string associated with the first positional parameter, $1, is the name of a directory. If so, it runs an **ls** command to display the contents of the directory.

Any number of commands may come between the **then** and the **endif** that ends the control structure. For example, you might have written:

```
if ( -d $1 ) then
     echo "$1 is a directory"
     ls $1
 endif
```

This example also shows that the commands do not have to begin on the same line as **then**, and the condition being tested does not have to begin on the same line as **if**. The condition and the commands are indented to make them stand out more clearly. This is a good way to make your shell scripts easier to read.

Another form of the **if** conditional is:

```
if (expr) then
commands
else
commands
endif
```

If the condition is true, the commands after the **then** are run; otherwise, the commands after the **else** are run. For example, suppose you know that the string associated with the variable *pathname* is the name of either a directory or a file. Then you could write:

```
if (  -d $pathname ) then
     echo "$pathname is a directory"
     ls $pathname
 else
     echo "$pathname is a file"
     cat  $pathname
 endif
```

If the value of *pathname* is the name of a file, this shell script uses **echo** to display an appropriate message, and then uses **cat** to display the contents of the file.

The final form of the **if** control structure is:

```
if (expr1) then
commands1
else if (expr2) then
commands2
else if (expr3) then
commands3
else
commands
endif
```

In this example, if *expr1* is true, *commands1* are run; otherwise, the shell goes on to check *expr2*. If that is true, *commands2* are run; otherwise, the shell goes on to check *expr3* and so on. If none of the test conditions are true, the *commands* after the **else** are run. Here is an example of how this can be used:

```
if ( !  $?argv ) then
     echo "no positional parameters"
 else if ( -d $1 ) then
     echo "$1 is a directory"
     ls $1
 else if ( -f $1 ) then
     echo "$1 is a file"
```

```
    cat  $1
else
    echo "$1 is just a string"
endif
```

The test after the **if** determines if the value of the first positional parameter, `$1`, is an empty string. If so, there are no positional parameters, so the shell script uses **echo** to display an appropriate message; otherwise, the script checks to see if the parameter is a directory name; if so, the contents of the directory are listed with **ls** (after an appropriate message). If that does not work, the script checks to see if the parameter is a filename; if so, the contents of the file are listed with **cat** (after an appropriate message). Finally, if none of the previous tests work, the parameter is assumed to be an arbitrary string, and the script displays a message to this effect.

You could put that script into a file named **listit** and run commands of the form:

```
listit name
```

to list the contents of *name* in a useful form.

The while Loop

The **while** loop repeats one or more commands while a particular condition is true. The loop has the form:

```
while (expr)
commands
end
```

The shell first tests to see if *condition* (expr) is true. If it is, the shell runs the *commands*. The shell then goes back to check the *condition*. If it is still true, the shell runs the *commands* again, and so on, until the *condition* is found to be false.

As an example of how this can be used, suppose you want to run a program named **prog** 100 times to get an idea of the program's average running speed. The following shell script does the job:

```
@ i=100
date
while ( $i > 0)
    prog
    @ i--
end
date
```

The script begins by setting a variable *i* to 100. It then uses the **date** command to get the current date and time.

Next the script runs a **while** loop. The condition says that the loop should keep on going as long as the value of *i* is greater than zero. The commands of the loop run **prog** and then subtract 1 from the *i* variable, similar to C programming language syntax. In this way, *i* goes down by 1 each time through the loop, until it is no longer greater than 0. At this point, the loop stops and the final instruction of the script prints out the date and time at the end of the loop. The difference between the starting time and the ending time should give some idea of how long it took to run the program 100 times.

(Of course, the shell itself takes some time to perform the condition and to do the calculations with *i*. If **prog** takes a long time to run, the time spent by the shell is relatively unimportant; if **prog** is a quick program, the extra time that the shell takes may be large enough to make the timing incorrect.)

The foreach Loop

The final control structure to be examined is the **foreach** loop. It has the form:

```
foreach name (wordlist)
commands
end
```

The parameter *name* should be a variable name; if this variable doesn't exist, it is created. The parameter *list* is a list of strings separated by spaces. The shell begins by assigning the first string in *list* to the variable *name*. It then runs the *commands* once. Then the shell assigns the next string in *list* to *name*, and repeats the *commands*. The shell runs the *commands* once for each string in *list*.

As a simple example of a shell script that uses **foreach**, consider:

```
foreach file ( *.c )
  c89 $file
end
```

When the shell looks at the **foreach** line, it expands the expression `*.c` to produce a *list* containing the names of all files (in the working directory) that have the suffix **.c**. The variable *file* is assigned each of the names in this list, in turn. The result of the **foreach** loop is to use the **c89** command to compile all **.c** files in the working directory. You could also write:

```
foreach file ( *.c )
  echo $file
  c89 $file
end
```

so that the shell script displayed each filename before compiling it. This would let you keep track of what the script was doing.

As you can see, the **foreach** loop is a powerful control structure. The *list* can also be created with command substitution, as in:

```
foreach file ( `find . -name "*.c" -print` )
     echo $file
     c89 $file
end
```

Here the **find** command finds all **.c** files in the working directory, and then compiles these files. This is similar to the previous shell script, but also looks at subdirectories of the working directory.

Combining Control Structures

You can combine control structures by nesting (that is, putting one inside another). For example:

```
foreach file ( `find . -name "*.c" -print` )
        if ( -M $file > -M $1 ) then
                echo $file
                c89 -c $file
        endif
end
```

This shell script takes one positional parameter, giving the name of a file. The script looks in the working directory and finds the names of all **.c** files. The **if** control structure inside the **foreach** loop tests each file to see if it is older than the file named on the command line. If the **.c** file is older, **echo** displays the name, and the file is compiled. You can think of this as making a set of files up to date with the filename specified on the command line.

Chapter 10. Using Job Control in the Shells

When you enter a shell command, you start a *process,* the execution of a function. When you enter that command, the shell runs it in its own *process group*. As such, it is considered a separate *job* and the shell assigns it a *job identifier*—a small number known only to the shell. (A shell job identifier identifies a shell job, not an MVS job.) When the process completes, the system displays the shell prompt.

The system also assigns a *process group identifier (PGID)* and a *process identifier (PID)*. When only one command is entered, the PGID is the same as the PID. The PGID can be thought of as a systemwide identifier. If you enter more than one command at a time using a pipe, several processes, each with its own PID, will be started. However, these processes all have the same PGID and shell job identifier. The PGID is the same as the PID of the first process in the pipe.

To sum it up, there are several types of process identifiers associated with a process:

PID A process ID. A unique identifier assigned to a process while it runs. When the process ends, its PID is returned to the system. Each time you run a process, it has a different PID (it takes a long time for a PID to be reused by the system). You can use the PID to track the status of a process with the **ps** command or the **jobs** command, or to end a process with the **kill** command.

PGID Each process in a process group shares a process group ID (PGID), which is the same as the PID of the first process in the process group. This ID is used for signaling related processes.

If a command starts just one process, its PID and PGID are the same.

PPID A process that creates a new process is called a *parent process*; the new process is called a *child process*. The parent process ID (PPID) becomes associated with the new child process when it is created. The PPID is not used for job control.

Several job control commands can either take as input or return the job identifier, process identifier, or process group identifier: **bg**, **fg**, **jobs**, **kill**, and **wait**.

The **nice** and **renice** commands can be used to change the priority of processes. Their use is dependent on the way performance groups have been prioritized at your installation; check with your system administrator for information about using **nice** and **renice** to change job priority.

Running Several Jobs at Once (Foreground and Background)

The can run more than one job at a time. While one is running in the foreground, one or more can be running in the background.

After you enter a command, you see the output from the command displayed on your screen. You cannot enter any other commands until the shell prompt ($ or >) appears. This command has run as a *foreground job*. Commands that take a few seconds to complete are convenient to run in the foreground.

You may prefer to run as *background jobs* those shell commands that take longer to run, because they prevent you from running any other commands while they are running in the foreground. The shell does *not* wait for the completion of a

background command before returning a prompt to you. Instead, while the command runs in the background, you can continue entering other commands on the command line.

In TSO/E, a *background job* is one that is typically entered at a workstation by a SUBMIT command. Like a TSO/E background job or an MVS batch job, an OS/390 UNIX *background job* runs without user interaction.

You can run a shell background job by any of these methods:

- Start the job in the background when you first enter it.
- Move a job from the foreground to the background.
- Use JCL with BPXBATCH. This utility is discussed in "The BPXBATCH Utility" on page 161.

Starting a Job in the Background with an Ampersand (&)

To start a command as a background job, simply end the command line with an ampersand (&). For example:

```
sort myfile >myout &
```

When the background job starts to run, the system:

- Assigns it a job identifier, a process group ID (PGID), and a process ID (PID)
- Displays the job identifier (in brackets) and one or more PIDs (more than one if there is a pipe)
- Then issues the shell prompt so that you can enter another command.

The first (or only) PID is also the PGID. This is an example of the output when you enter a background command:

```
$sort myfile >myout &
[3] 717046
$
```

3 is the job identifier and 717046 is the PID and PGID.

Note the PID numbers and the job number when you create a background job; you can use them to check the status of the job or to end it.

Unlike MVS batch jobs, a shell job running in the background directs its output to standard output, your workstation screen. If you do not want to have this output interfering with your work in the foreground, remember to redirect the output to a file when you start a background command. After the output is redirected, you can look at it whenever it is convenient.

A background job can be suspended. A background job that attempts to read from **stdin** is suspended until it is made the foreground process. Therefore, if a program reads from **stdin**, you may want to redirect **stdin** from a file. Also, if the `tostop` setting of the terminal is enabled (you can set or query this by using the **stty** command), output from a background job causes the job to be suspended.

Moving a Job to the Background

Suppose you want to move the foreground job to the background, where it can run while you enter other commands in the foreground. To put the job in the background:

1. Stop the job by entering <EscChar-Z>. A message displays the job identifier.

2. Enter the **bg** command. You may need to specify the job identifier with **bg** if there is more than one stopped job. If you do not specify a job identifier, **bg** uses the most recently stopped job.

A message displays the job identifier and the command that is running in the background.

Moving a Job to the Foreground

When you want to move a job from the background to the foreground, use the **fg** command. If there are multiple background jobs, you need to supply the job identifier preceded by a % sign. For example:

```
fg %7
```

Checking the Status of Jobs

You can use either the **jobs** command or the **ps** command to check on the status of jobs.

Using the jobs Command

The **jobs** command reports the status of background processes currently running, based on the job identifier; it reports on the status of stopped processes and completed processes also. If you use the **-l** option, you can display both the job identifier and the PID for the process.

Say you entered a command that involves more than one process—for example:

```
myprog | grep write
```

If you want to check the status of that command, use the **jobs -l** command. The status message displays the job identifier, the PID number for each process in the job, the status of the command, and the command being run; in this case the status message shown in the OS/390 Shell is:

```
[1] 720902 + Stopped (SIGTSTP)  myprog|grep write
  720902        alive           -sh
  458759        alive           -sh
```

In this case:

- The job identifier is 1 (from [1]).
- The PIDs of the processes are 720902 and 458759.
- The PGID is 720902 (the PID of the first process in the process group).

The status message for the tcsh shell is similar to the example given above.

Using the ps Command

You can use the **ps** command to display a list of your processes that are currently running and obtain additional information about those processes. (Only a superuser or a user with appropriate permissions can obtain information about all processes.)

For example, here the **ps** command displays the status of started processes:

```
    PID TTY       TIME COMMAND
 262148 ttyp0000  2:46 /bin/sh
 196614 ttyp0000  0:22 ./myprog
  65543 ttyp0000  0:13 /bin/grep
 196616 ttyp0000  2:07 /bin/ps
```

PID This is a PID displayed as decimal value.

TTY The name of the controlling terminal, if any. The *controlling terminal* is the

workstation that started the process. On a system with more than one workstation, the names of the workstations that have started processes are listed here.

TIME The amount of central processor time the process has used since it began running.

COMMAND
The name of the command or program that started the process. The display indicates which directory the command or program is found in. For example, the **ps** command is in **/bin**.

Usually, just issuing **ps** will tell you all you need to know about your current processes. However, there are a number of options you can use to tailor the displayed information. For example, you can use the **-a** option to display only processes associated with a terminal, not the system processes. Read the command description of **ps** in *OS/390 UNIX System Services Command Reference*.

Canceling a Job

Often you will start a job and then decide to interrupt it before it completes. You can do this regardless of whether the job is running in the foreground or background.

Canceling a Foreground Job

To cancel a foreground job, enter <EscChar-C>. The command stops and the shell displays the shell prompt.

Canceling a Background Job

To cancel a background job, use the **kill** command. To be able to kill a process, you must own it. (The superuser, however, can kill any process except **init**.)

Before you can cancel a background job, you need to know either a PID, job identifier, or PGID. You can use the **jobs** command to determine any of these.

The format of the **kill** command in the OS/390 shell is:

```
kill [-s signal name] [pid] [job-identifier]
```

The format of the **kill** command in the tcsh shell is:

```
kill [-signal name] [pid] [job-identifier]
```

To kill one process, use its PID. For example:

```
kill 717
```

would kill the process with the PID 717. Any other processes in the job—from a pipe—would not be killed.

To kill a particular process group, you can use a job identifier or a negative PGID.

- You can use the job identifier for one process in the group preceded with a % to kill every process in the group. In the OS/390 shell, use:

  ```
  kill -s KILL %7
  ```

  In the tcsh shell, use:

  ```
  kill -KILL %7
  ```

- You can use a negative PGID to kill every process in a process group. (As mentioned earlier, the PGID is the PID for the first process in the process group.) For example, in the OS/390 shell:

  ```
  kill -s KILL -- -123456
  ```

  will kill every process in the process group with PGID 123456.

  In the tcsh shell:

  ```
  kill -KILL -123456
  ```

  will kill every process in the process group with PGID 123456.

Stopping and Resuming a Job

Occasionally, you may want to stop a job that is running in the foreground or background, perform a different task, and then later resume the stopped job.

Stopping a Foreground Job

To stop a foreground job, enter <EscChar-Z>. A message displays the job identifier, the status `Stopped`, and the command that is stopped.

Stopping a Background Job

To stop a background job, use the **kill** command with the STOP signal and the job identifier preceded with a `%`. For example, in the OS/390 shell:

```
kill -s -STOP %3
```

stops the background job with the job identifier 3.

```
kill -STOP %3
```

does the same in the tcsh shell.

Resuming a Stopped Job

When you are ready to resume a stopped job, you can resume it in the foreground using the job identifier. Enter:

```
fg %n
```

where *n* is the job identifier for the stopped job.

To resume a stopped job in the background, enter:

```
bg %n
```

where *n* is the job identifier for the stopped job. The `%n` is unnecessary if there is only one job.

Delaying a Command

If you want to delay a command from running until a previous background job has completed, you can use the **wait** command. You need to know the job identifier of the job you want to wait for; you can use the **jobs** command to get that. For example, the command:

```
wait %7; print "Time for tea"
```

means that "`Time for tea`" will display on your screen when the command whose job identifier is 7 finishes running.

Exiting the Shell with Background Jobs Running

When you exit the shell, any stopped background jobs are terminated. But if you have a background job in the running state, you can exit the shell without terminating it.

In the OS/390 shell, the default setting **set -m** runs background jobs in a separate process group. Jobs in a separate process group are not sent a **SIGHUP** signal when you exit the shell. With the default **-m** setting, background jobs continue to run after you exit the shell.

In the tcsh shell, use **NOHUP** to exit the shell with background jobs running.

OMVS Interface

To exit with a background job running, use the **quit** subcommand. (Type **quit** and press the Subcommand function key or switch to subcommand mode and enter the **quit** command.) A background job that is running will continue running.

If you are using the OMVS interface and you use the **exit** command to exit the shell while you have a shell background job running, OMVS may send this message:

```
The shell process ended, but the session did not end automatically.
You may need to run the QUIT subcommand to end the session.
```

Asynchronous Terminal Interface

To exit when a background job is running, type <Ctrl-D> or use the **exit** command. A background job that is running will continue running. You do not get any indication that a background job is running.

Changing the Default in the OS/390 Shell

If you change the setting to **+m**, background jobs end when you exit the shell. If you have changed the setting to **+m** and you want to start a long-running command and have it continue running after you exit the shell, use the **nohup** command and an ampersand (**&**):

```
nohup 'command-line' &
```

For example:

```
nohup sort -u file1 >output 2>>outerr &
```

Ending the **nohup** command with an **&** makes the command run in the background, even after you exit the shell.

Deciding How to Submit Background Jobs

Table 2 on page 157 compares two methods for submitting a background job:

- Typing an `&` (ampersand) after the shell command
- Using JCL with BPXBATCH. This utility is discussed in "The BPXBATCH Utility" on page 161.

Table 2. Comparison of Running a Background Job from the Shell or from MVS

Topic	Shell (command &)	MVS (JCL with BPXBATCH)
Starting the job	Background jobs start running immediately.	Background jobs are put in a queue; there may be a wait until the job starts running.
Interactive access	You can see output from the job displayed on the terminal. You can move the job to the foreground if you need to give it input, and then move it to the background again.	Background jobs run separately; you cannot interact with them. However, if you redirect output to a file in the file system, from your interactive shell session you could periodically browse the output file to see what is in it. You could do this with any of these commands: **cat**, **pg**, **more**, **obrowse**, or the TSO/E OBROWSE command.
System limits	Due to system limits on the number of processes per user, multiple background jobs run by the same user could fail at some point.	Due to system limits on the number of processes per user, multiple background jobs run by the same user could fail at some point.
Managing the job	You can use **ps**, **kill**, **bg**, **fg** and **jobs** on the background job.	You can use **ps** and **kill** on the background job.
Impact on System	Creates an immediate demand on the system to support another address space. This could degrade performance for all users.	The system determines when it is a reasonable time to run batch jobs. Batch work can be suspended during periods of heavy interactive workload.

Chapter 11. Using OS/390 UNIX System Services from Batch, TSO/E, and ISPF

Note: Information in this chapter is directed towards users of the OS/390 shell. Most examples pertain to the OS/390 shell and not the tcsh shell.

You can access OS/390 UNIX services from MVS Batch, TSO/E, or ISPF, using:

- Job control language (JCL) to run shell scripts or OS/390 UNIX application programs as MVS batch (background) jobs. This chapter describes the JCL that supports the hierarchical file system (HFS). For more general information about JCL, see *OS/390 MVS JCL User's Guide* and *OS/390 MVS JCL Reference*.
- BPXBATCH, an MVS utility, to
  - Run shell scripts and executable files in MVS batch. An *executable file* is an OS/390 C/C++ application program that has been compiled and link-edited and resides in the hierarchical file system (HFS).
  - Run shell commands, shell scripts, and executable files from the TSO/E `READY` prompt.
- TSO/E commands designed to work with MVS. See "Using Commands to Work with Directories and Files" on page 201 and also Chapter 20 for more information. For the complete command descriptions, see the section on TSO/E commands in *OS/390 UNIX System Services Command Reference*.
- REXX programs written using OS/390 UNIX extensions called *syscall commands*.
- The ISPF shell.

Note: When you submit a shell script or OS/390 UNIX application program as an MVS batch job, the job has a foreground process group ID. (This is because a process started *without* a trailing & is handled as a foreground process by the OS/390 shell.) If the process is canceled, a **SIGHUP** signal is sent to the process group.

JCL Support for OS/390 UNIX System Services

JCL data definition (DD) statements use a *data definition name (ddname)* to specify the data to be used by the program that you are submitting as a batch job. The ddname is used in two places:

1. In your application program. Here the ddname refers to nonspecific data, rather than a specific data set name or pathname.
2. In the JCL used to submit the application program as a background job. Here it "binds" the nonspecific reference in the program to a specific data set name or pathname.

You can specify an HFS file in the JCL for user-written applications or for IBM-supplied services, such as:

- The Data Facility System-Managed Storage/MVS (DFSMS/MVS) program management binder, a prelinker and linkage editor
- BPXBATCH
- The TSO/E OCOPY command

Note: In this discussion, references to JCL also apply to the equivalent dynamic allocation functions.

The PATH Keyword

You can use the PATH keyword on a DD statement to specify the pathname for an HFS file. When the PATH keyword is used in JCL, you can also usethese keywords:

- **PATHOPTS** to indicate the access for the file (for example, read or read-write) and to set the status for the file (for example, append, create, or truncate). This is analogous to the C **open()** function's option arguments.

  Note: If you specify either OCREAT or OCREAT together with OEXCL on the PATHOPTS parameter and the file does not exist, OS/390 UNIX performs an **open()** function. The options from PATHOPTS, the pathname from the PATH parameter, and the options on PATHMODE (if specified) are specified in the **open()**. OS/390 UNIX uses the **close()** function to close the file before the application program receives control.
- **PATHMODE** to indicate the permissions, or file access attributes, to be set when a file is being created. This is analogous to the **open()** function's mode arguments.
- **PATHDISP** to indicate how MVS should handle the file when the job step ends normally or abnormally. This performs the same function as the DISP parameter for a data set.

If PATHOPTS and PATHMODE are absent from the DD statement, an application needs to supply defaults for the options and mode, or issue an error message and fail.

The DSNTYPE Keyword

There are two related subparameters on the DSNTYPE keyword of theDD statement:

- HFS (hierarchical file system)
- PIPE (named pipe)

For more information on the JCL keywords, see *OS/390 MVS JCL Reference*.

Using the ddname in an Application

Instead of using data set names or pathnames in an application, you can use a ddname; then in the JCL, you associate a specific data set or file with that ddname.

Note: The parent process's allocations, for both data sets and files, are not propagated by **fork()** and are lost on **exec**, except for STEPLIB.

You have a choice of two methods for accessing data sets and files in an application:

- The ANSI C function **fopen()**
- The OPEN macro

The fopen() Function

The **fopen()** functionrecognizes and handles the difference between a ddname associated with a data set (DSN keyword) or a pathname (PATH keyword). For example:

```
fopen("dd:FRED", "r+")
```

takes the ddname FRED, determines if FRED refers to a ddname for a file or a data set, and opens it. Once a file is opened, **fread()** and **fwrite()** can access the data.

The OPEN Macro

The OPEN macro can open an HFS filespecified with the PATH keyword or a data set specified with the DSN keyword. The macro supports DD statements that specify the PATH parameter only for data control blocks that specify DSORG=PS (note that EXCP is not allowed). DFSMSdfp supports BSAM and QSAM interfaces tothese types of HFS files: regular files, character special files (null files only), FIFO special files, and symbolic links. You cannot open directories or external links.

For more information on BSAM and QSAM interface support for access to HFS files, see *OS/390 DFSMS Macro Instructions for Data Sets*.

Specifying a ddname in the JCL

In the JCL for a job, you associate a ddname with the name of a specific data set or file.

To specify a file, you use the PATH keyword. For example:

```
//FRED DD  PATH='/u/fred/list/wilma'
```

associates the pathname for the file **/u/fred/list/wilma** with the ddname FRED. At another time, you might specify a different file to be associated with the ddname FRED.

To specify a data set, you use the DSN keyword. For example:

```
//FRED DD  DSN=FRED.LIST.WILMA,DISP=SHR
```

associates the data set FRED.LIST.WILMA with the ddname FRED. At another time, you might specify a different data set to be associated with the ddname FRED.

The BPXBATCH Utility

BPXBATCH is an MVS utility that you can use to run shell commands or shell scripts and to run executable files through the MVS batch environment. You can invoke BPXBATCH:

- In JCL
- From the TSO/E `READY` prompt
- From TSO CLISTs and REXX execs
- From a program

BPXBATCH has logic in it to detect when it is run from JCL. If the BPXBATCH program is running as the only program on the job step task level, it sets up the **stdin**, **stdout**, and **stderr** and execs the requested program. If BPXBATCH is not running as the only program at the job step task level, the requested program will run as the second step of a JES batch address space from JCL in batch. If run from any other environment, the requested program will run in a WLM initiator in the OMVS subsys category.

BPXBATSL is provided as an alias for BPXBATCH. BPXBATSL performs a local spawn but does not require resetting of environment variables. BPXBATSL behaves exactly like BPXBATCH, and allows local spawning whether the current environment is set up or not.

This book provides examples of how you can use BPXBATCH. For more detailed information about BPXBATCH, see *OS/390 UNIX System Services Command Reference*.

Defining Standard Input, Output, and Error for BPXBATCH

OS/390 c/c++ programs require that **stdin**, **stdout**, and **stderr** be defined as either a file or a terminal. Many C functions use **stdin**, **stdout**, and **stderr**. For example, **getchar()** obtains a character from **stdin**, **printf()** directs the output to **stdout**, and **perror()** directs the output to **stderr**. See “Understanding Standard Input, Standard Output, and Standard Error” on page 68 for more information.

For BPXBATCH, the default for **stdin** and **stdout** is **/dev/null**. The BPXBATCH default for **stderr** is the same file defined for **stdout**. For example, if you define **stdout** to be **/tmp/output1** and do not define **stderr**, then both **printf()** and **perror()** output is directed to **/tmp/output1**.

For BPXBATCH, you can define **stdin**, **stdout**, and **stderr** using one of these:

- The TSO/E ALLOCATE command, using the ddnames STDIN, STDOUT, and STDERR.
- A JCL DD statement with the PATH operand, using the ddnames STDIN, STDOUT, and STDERR.
- Redirection. For example, even if **stdout** defaults to **/dev/null**, the command:

  ```
  BPXBATCH SH ps -el >>/tmp/ps.out
  ```

  entered in TSO/E redirects the output of the **ps** command to be appended to the file **/tmp/ps.out**.

Defining an Environment Variable File for BPXBATCH

When you are using BPXBATCH to run a program, you typically pass the program a file that sets the environment variables. If you do not pass an environment variable file when running a program with BPXBATCH or if the **HOME** and **LOGNAME** variables are not set in the environment variable file, those two variables are set from your logon RACF profile. **LOGNAME** is set to the user name and **HOME** is set to the initial working directory from the RACF profile.

To pass environment variables to BPXBATCH, you define a file containing the variable definitions; it can be one of these:

- An HFS file identified with a STDENV statement
- An MVS data set identified with a STDENV statement

The default is **/dev/null**.

If you define an HFS file:

- It must be a text file defined with read access only.
- Specify one variable per line, in the format `variable=value`. Environment variable names must begin in column 1.
- An environment variable file cannot have sequence numbers in it. If you use the ISPF editor to create the file, set the sequence numbers offby typing `number off` on the command line before you begin typing the data. If sequence numbers already exist, type `UNNUM` to remove them and set number mode off.

If you define an MVS data set:

- It must be a sequential data set, a partitioned data set (PDS) member, or a SYSIN data set. Record format can be fixed or variable (nonspanned).
- Specify one environment variable per record, in the format `variable=value`. Environment variable names must begin in column 1. Do not use terminating nulls.

- An environment variable file cannot have sequence numbers in it. If you use the ISPF editor to create the file, set the sequence numbers off by typing `number off` on the command line before you begin typing the data.

You can specify the environment variables as part of an in-stream JCL SYSIN data set—for example:

```
//STDENV DD *
variable1=aaaaaaa
variable2=bbbbbbbb
.
.
variable5=ffffffff
/*
```

Note: Trailing blanks are truncated for SYSIN data sets, but not for other data sets.

For BPXBATCH, you can specify the environment variable file by using one of these:

- The TSO/E ALLOCATE command—for example:

  ```
  ALLOCATE DDN(STDENV) DSN('TURBO.ENV.FILE') SHR
  ```

- A JCL STDENV DD statement. To identify an HFS file, use the PATH operand and the PATHOPTS ORDONLY. For example:

  ```
  //STDENV DD PATH='u/turbo/env.file',PATHOPTS=ORDONLY
  ```

- JCL with a SYSIN data set for the environment variable definitions.
- SVC 99 dynamic allocation, if you are running BPXBATCH from a program.

Example: Setting Up Code Page Support in a STDENV file

To enable national language support for BPXBATCH, set the locale variables in the STDENV file. For example, you could put these lines in the file:

```
LANG=Da_DK.IBM-277
LC_ALL=Da_DK.IBM-277
```

After you allocate this file to STDENV, you can test it by typing:

```
OSHELL echo $HOME
```

The pathname of your home directory should be displayed, instead of just **$HOME**.

_BPX_BATCH_SPAWN and _BPX_BATCH_UMASK Environment Variables

BPXBATCH uses two environment variables for execution that are specified by STDENV:

- **_BPX_BATCH_UMASK=0755**
- **_BPX_BATCH_SPAWN=YES|NO**

_BPX_BATCH_UMASK allows the user the flexibility of modifying the permission bits on newly created files instead of using the default mask (when PGM is specified).

Note: This variable will be overridden by **umask** (usually set from within /etc/profile) if BPXBATCH is invoked with the 'SH' option (SH is the default). SH causes BPXBATCH to execute a login shell which runs the /etc/profile script (and runs the user's .profile) and which may set the umask before execution of the intended program.

_BPX_BATCH_SPAWN causes BPXBATCH to use spawn instead of fork/exec and allows data definitions to be carried over into the spawned process. When **_BPX_BATCH_SPAWN** is set to YES, spawn will be used. If it is set to NO, which is equivalent to the default behavior, fork/exec will be used to execute the program.

If **_BPX_BATCH_SPAWN** is set to YES, then you must consider two other environment variables that affect spawn (BPX1SPN):

- **_BPX_SHAREAS=YES|NO|REUSE**

  When YES or REUSE, the child process created by spawn will run in the same address space as the parent's under these conditions:

  – The child process is not setuid or setgid to a value different from the parent
  – The spawned filename is not an external link or a sticky bit file
  – The parent has enough resources to allow the child process to reside in the same address space
  – The NOSHAREAS extended attribute is not set

  When NO, the child and parent run in separate address spaces.
- **_BPX_SPAWN_SCRIPT=YES**

  When YES, the spawn will treat the specified file as a shell script and will invoke the shell to run the shell script.

Setting **_BPX_SPAWN_SCRIPT=YES** improves shell script performance. See "Improving the Performance of Shell Scripts" on page 44 for more information. For more information about spawn, see BPX1SPN in *OS/390 UNIX System Services Programming: Assembler Callable Services Reference*.

Invoking BPXBATCH in JCL

You can create a job that uses BPXBATCH to run a shell command, a shell script, or an executable file.

BPXBATCH is invoked in JCL in this way:

```
//stepname EXEC PGM=BPXBATCH,PARM='SH|PGM program_name'
```

where:

- When `SH` is specified, `program_name` is the name of a shell command or a file containing a shell script. `SH` is the default; therefore, you can omit `PARM=` and supply the name of a shell script for STDIN, and it will be invoked.
- When `PGM` is specified, `program_name` is the name of an executable file or a REXX exec that is stored in an HFS file.

If you specify data sets in a STEPLIB DD statement, all the data sets must be cataloged.`UNIT=` and `VOL=SER=` parameters are not propagated to the process that is being executed by BPXBATCH.

If the job needs to run with a group other than your default group, on the job card you need to code `GROUP=grpname` to specify the group your job needs to run under. For BPXBATCH, the group needs to have an OMVS segment and a GID defined for it.

If your job requires a REGION size greater than the default onyour system, you may receive this abend code:

```
ABEND 4093 reason code 0000001c
```

To fix this, use a larger REGION size. For example, you could specify:

```
//SHELLCMD EXEC PGM=BPXBATCH,REGION=8M,PARM='SH shell_cmd'
```

Running a Shell Script in Batch

You can use BPXBATCH to run a shell script through MVS batch and redirect the output error messages to an HFS file. Because the default is `PARM='SH'`, the `PARM` is not specified in the following example. The shell script associated with STDIN is invoked. You can allocate STDIN, STDOUT, and STDERR as files, using the PATH operand. In this example, for user TURBO:

- The pathname of the STDIN file (the shell script) is **/u/turbo/bin/myscript.in**
- The pathname of the STDOUT file is **/u/turbo/bin/mystd.out**
- The pathname of the STDERR file is **/u/turbo/bin/mystd.err**

```
//jobname  JOB ...
//stepname EXEC PGM=BPXBATCH,REGION=8M
//STDIN    DD  PATH='/u/turbo/bin/myscript.in',PATHOPTS=(ORDONLY)
//STDOUT   DD  PATH='/u/turbo/bin/mystd.out',PATHOPTS=(OWRONLY,OCREAT),
//             PATHMODE=SIRWXU
//STDERR   DD  PATH='/u/turbo/bin/mystd.err',PATHOPTS=(OWRONLY,OCREAT),
//             PATHMODE=SIRWXU
```

Running a Shell Command in Batch

In the following example, BPXBATCH runs the shell command **compress** to compress the file /usr/lib/junk. To start the next JCL job step before the **compress** command completes, PARM is specified as:

```
SH nohup compress /usr/lib/junk & sleep 1
```

If the same job used PARM=″SH compress /usr/lib/junk″, the job step waits for the **compress** shell command to end. For short-running commands, this is fine. But for long-running commands, where you want to use BPXBATCH to start a shell command in the background and not wait for completion, you must follow the example:

```
PARM="SH nohup compress /usr/lib/junk & sleep 1"
```

SH starts a ″login shell″ to parse and run the command input. The login shell parses the ″&″, signifying that the command is to run asynchronously (in the background), and forks a child process to run the **nohup** command. In the child process, the **nohup** shell command (which takes another command as an argument), prevents the process from being terminated when the login shell returns to BPXBATCH. In parallel with the **nohup** processing, the login shell runs the **sleep** command. Running the **sleep** command delays the login shell from returning to BPXBATCH until the child process has had enough time (1 second) to protect itself from being terminated. The login shell returns to BPXBATCH, while the child process continues to run the **compress** command.

In this example, STDERR is an HFS file allocated using the PATH operand. The pathname of the standard error (STDERR) file is /u/turbo/bin/mystd.err.

```
//jobname  JOB ...
//stepname EXEC PGM=BPXBATCH,REGION=8M,PARM='SH nohup compress /usr/lib/junk & sleep 1'
//STEPLIB DD DSN=CEE.SCEERUN,DISP=SHR
//STDERR   DD  PATH='/u/turbo/bin/mystd.err',
//             PATHOPTS=(OWRONLY,OCREAT,OTRUNC),PATHMODE=SIRWXU
```

The STEPLIB is propagated for the execution of the shell and for any processes created by the shell.

Invoking BPXBATCH from TSO/E READY

You can use BPXBATCH to run a shell command or a shell script from TSO/E, but it is even easier to use the OSHELL exec, which invokes BPXBATCH.

The syntax for using BPXBATCH to run a shell command from TSO/E is:

```
BPXBATCH SH shellcmd
```

- SH starts a login shell which processes your .profile before running a command or shell script.
- **shellcmd** is the name of a shell command or shell script.

OSHELL: Running a Shell Command from TSO/E READY

The OSHELL REXX exec, shipped in SYS1.SBPXEXEC, invokes BPXBATCH to run non-interactive shell commands from the TSO/E `READY` prompt. The output is displayed in your TSO/E session.

OSHELL Usage Notes:

1. With OSHELL, you cannot use a shell command with an & (ampersand) to run it in the background.
2. OSHELL cannot be used to invoke an interactive shell command.
3. OSHELL creates a temporary file in the **/tmp** directory. The name of the temporary file includes the time, to avoid naming conflicts (for example, **/tmp/userid1.12:33:32.461279.IBM**). The file is deleted when OSHELL completes.

OSHELL Examples:

For example, to delete the file **dbtest.c**, at the TSO/E `READY` prompt, user TURBO would enter:

```
oshell rm -r /u/turbo/testdir/dbtest.c
```

To display the amount of free space in your file system, you could enter:

```
oshell df -P
```

To display information on all accessible processes, you could enter:

```
oshell ps -ej
```

Figure 18 on page 167 shows how OSHELL is coded.

```
/* REXX */
parse arg shellcmd
username =,
TRANSLATE(userid(),'abcdefghijklmnopqrstuvwxyz','ABCDEFGHIJKLMNOPQRSTUVWXYZ')
/*********************************************************************/
/* Free STDERR just in case it was left allocated                    */
/*********************************************************************/
/*                                                                   */
msgs = msg('OFF')
"FREE DDNAME(STDERR)"
/*********************************************************************/
"ALLOCATE FILE(STDOUT) PATH('/tmp/"username"."time('L')".IBM') ",
"PATHOPTS(OWRONLY,OCREAT,OEXCL,OTRUNC) PATHMODE(SIRWXU)",
"PATHDISP(DELETE,DELETE)"
IF RC ¬= 0 Then
  DO
    "FREE DDNAME(STDOUT)"
    "ALLOCATE FILE(STDOUT) PATH('/tmp/"username"."time('L')".IBM') ",
    "PATHOPTS(OWRONLY,OCREAT,OEXCL,OTRUNC) PATHMODE(SIRWXU)",
    "PATHDISP(DELETE,DELETE)"
    IF RC ¬= 0 Then
      DO
        msgs = msg(msgs)
        /* Allocate must have failed */
        Say ' This REXX exec failed to allocate STDOUT.'
        Say ' This REXX exec did not run shell command ' shellcmd
        RETURN
      END
  END
msgs = msg(msgs)

"BPXBATCH SH "shellcmd

IF RC ¬= 0 Then
   DO
     Say ' RC = ' RC
     Say ' '
   END
IF RC > 255 Then
   DO
     Say ' Exit Status = ' RC/256
     Say ' '
   END
IF (RC ¬= 254) & (RC ¬= 255) THEN
   DO
     "ALLOCATE FILE(out1) DA(*) LRECL(255) RECFM(F) REUSE"
     "OCOPY indd(STDOUT) outdd(out1) TEXT PATHOPTS(OVERRIDE)"
     "FREE DDNAME(out1)"
   END
"FREE DDNAME(STDOUT)"
```

Figure 18. OSHELL REXX Exec

Using TSO/E REXX for OS/390 UNIX System Services Processing

You can use a set of OS/390 UNIX extensions to TSO/E REXX—host commands and functions—to access kernel callable services. The OS/390 UNIX extensions, called *syscall commands*, have names that correspond tothe names of the callable services that they invoke—for example, **access**, **chmod**, and **chown**.

You can run a REXX program with OS/390 UNIX extensions from MVS, TSO/E, the shell, or a C program. The exec is not portable to an operating system that does not have OS/390 UNIX installed.

For more information about the REXX extensions that call OpenMVS services, see *OS/390 Using REXX and OS/390 UNIX System Services*.

Using the ISPF Shell

With the ISPF shell, a user or system programmer can use ISPF dialogs instead of shell commands to perform many tasks, especially those related to file systems and files. An ordinary user can use the ISPF shell to work with:

- Directories
- Regular files
- FIFO special files
- Symbolic links, including external links

You can also run shell commands, REXX programs, and C programs from the ISPF shell. The ISPF shell can direct **stdout** and **stderr**only to an HFS file, not to your terminal. If it has any contents, the file is displayed when the command or program completes.

Invoking the ISPF Shell

You can invoke the shell by:

- Typing the TSO/E ISHELL command. See “Entering a TSO/E Command” on page 202 for information on entering TSO/E commands in TSO/E, the shell, and ISPF.
- Selecting the ISPF shell from the ISPF menu, if a menu option is installed.

Working in the ISPF Shell

Figure 19 on page 169 is the main panel, which you see when you invoke the ISPF shell. At the top of the panel is the action bar, with seven choices:

File
Directory
Special file
Tools
File systems
Options
Setup
Help

When you select one of these choices, a pulldown panel with a list of actions is displayed.

```
 File  Directory  Special_file  Tools  File_systems  Options  Setup  Help
------------------------------------------------------------------------------
                                  ISPF Shell

Enter a pathname and do one of these:

    - Press Enter.
    - Select an action bar choice.
    - Specify an action code or command on the command line.

Return to this panel to work with a different pathname.
                                                                    More:    +
   /
   _____________________________________________________________
   _____________________________________________________________
   _____________________________________________________________



 (C) Copyright IBM Corp., 1993. All rights reserved.
Command ===> _____________________________________________________________
 F1=Help    F3=Exit      F5=Retrieve  F6=Keyshelp  F7=Backward  F8=Forward
F10=Actions F11=Command  F12=Cancel
```

Figure 19. ISPF Shell: The Main Panel

In the center of the panel, you see three lines. Here you can type the pathname of a file (a directory is a type of file) that you want to work with. It can be the name of an existing file or a new file that you are creating.

In the lower part of the panel, you see a command line. Here you can type an *action code*, a one-character code that specifies an action that you want to perform on the pathname you are working with. For example, `D` is the action code for "delete." (To familiarize yourself with the action codes, press <F1> on the main panel. On the help panel that is displayed, position your cursor under the highlighted words *action code* and press <F1>.)

Work in the ISPF shell is a two-step sequence:
1. Select an *object*—the pathname of a new or existing file.
2. Select an action for that object.

Selecting an Object

When you select an object, this pathname becomes the object of most subsequent actions until you change it or delete it. You can select an object in either of these ways:
- Specify the pathname on the main panel
- Select a file on a directory list panel

On the Main Panel

On the main panel, you can supply a pathname for a directory, a regular file, a FIFO special file, symbolic link, external link, or, if you are a superuser, a character special file.

If you type a pathname for a file that does not exist and press <Enter>, you see a popup panel from which you can:
- Create a directory
- Create a regular file using ISPF File Edit
- Create a regular file by copying another file into it
- Create a regular file by copying a data set into it

- Create a FIFO special file
- Create a symbolic link or an external link
- Create a hard link to a file

An external link is a type of symbolic link and is handled as such.

If you type the name of an existing file and press <Enter>, the system performs the default action for that file type. The default actions, which you can change using the **Options** pulldown, depend on the file type:

- **Directory.** List the files in the directory
- **Regular file.** Browse the file
- **Character special file.** Display its attributes
- **FIFO special file.** Display its attributes
- **Symbolic link.** Display its attributes

From a Directory List

From the **Directory** pulldown, you can select a directory list. All the files in a directory are displayed in a list; you can then select a file by typing a slash (/) or an s or an action code next to its name.

Selecting an Action

To select an action, you can do one of these:

- Specify an action code on the command line of the main panel.
- Select an action from an action bar pulldown panel.
- Specify an action on a directory list panel.

On the Command Line

If you type a pathname on the main panel and then type an action code on the command line and press <Enter>, you can perform the same file system tasks that are available from the pulldown panels.

To familiarize yourself with the action codes, press <F1> on the main panel. On the help panel that is displayed, position your cursor under the highlighted words *action code* and press <F1>.

Using the Action Bar

Supply a pathname on the main panel. To use the action bar, position your cursor under one of the choices and press <Enter>. You then see a pulldown panel with a list of actions; Figure 20 on page 171 shows what you would see if you selected the **Directory** pulldown.

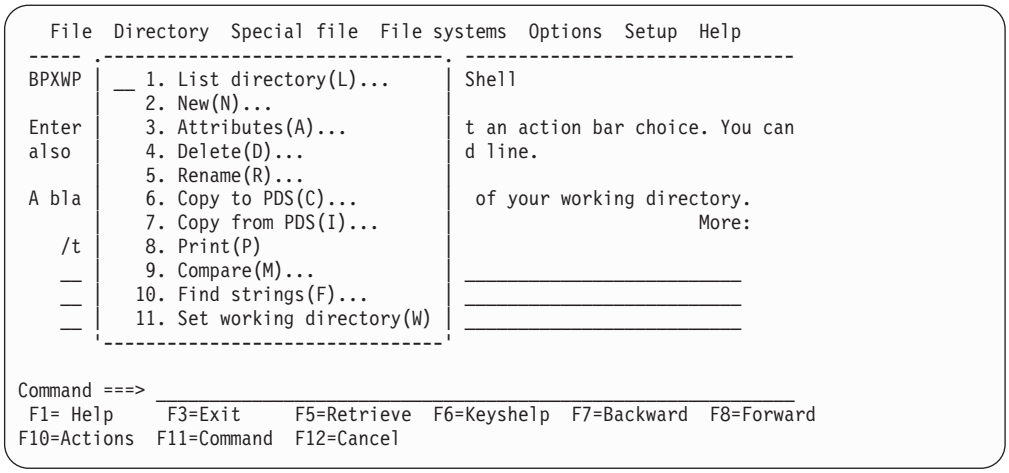


Figure 20. A Pulldown Panel Selected from the Action Bar

To select an action from the pulldown you can do either of these:
- Specify the number of the action you want to select and press <Enter>.
- Position the cursor next to the action and press <Enter>.

The letter in parentheses following each action is the action code for that action.

On a Directory List

From the **Directory** pulldown, you can select a directory list. All the files in a directory are displayed in a list; you can then select a file and specify an action by typing the action code next to the filename.

Using the Online Help Facility

In the ISPF shell, you can get help information for:
- Panels
- Fields on panels
- Highlighted words on panels

Position your cursor on one of those locations and press <F1>.

For more information on the online help facility when you begin working in the ISPF shell, select the `Help` choice on the action bar and read the information there.

Working with the File Pulldown

Using the **File** pulldown, you can perform the following tasks with a regular file.

Table 3. Using the ISPF Shell to Work with a Regular File

Task	Pulldown Choice
Create a file.	1. New(N)
Display and alter file attributes (such as permissions, owning user, owning group, audit settings).	2. Attributes(A)
Delete a file.	3. Delete(D)
Rename a file.	4. Rename(R)
Edit a file.	5. Edit(E)
Browse a file with fixed-length records.	6. Browse text(B)
Browse a text file that has delimited lines.	7. Browse records(V)

Table 3. Using the ISPF Shell to Work with a Regular File (continued)

Task	Pulldown Choice
Copy a file to another file.	8. Copy to(C)
Copy a file to a sequential data set, specifying one or both of these options: binary copy or code page conversion.	8. Copy to(C)
Replace the contents of a file with another file.	9. Replace from(I)
Replace the contents of a file with a sequential data set, specifying one or both of these options: binary copy and code page conversion.	9. Replace from(I)
Print a file to the ISPF list data set.	10. Print(P)
Compare a file with another file and put the results in an output file.	11. Compare(M)
Search a file for a specified text string.	12. Find Strings(F)
Run executable files.	13. Run(X)
Create a hard link to a file.	14. Link
Display the attributes for the file system the file is in.	15. File System (U)

Working with the Directory Pulldown

Using the **Directory** pulldown, you can perform the following tasks with a directory. To customize the way a directory list is sorted or displayed, use the **Options** pulldown.

Table 4. Using the ISPF Shell to Work with a Directory

Task	Pulldown Choice
List all files and subdirectories in a directory.	1. List directory(L)
Create new subdirectories within a directory.	2. New(N)
Display and alter directory attributes (such as permissions).	3. Attributes(A)
Delete a directory.	4. Delete(D)
Rename a directory.	5. Rename(R)
Copy all files in a directory to a partitioned data set or a PDS/E. If desired, specify one or more of these options: binary copy, inclusion of files with lowercase names, code page conversion, or stripping the suffix from the filenames.	6. Copy to PDS(C)
Copy all or selected members from a partitioned data set or PDS/E into a directory. If desired, specify one or more of these options: binary copy, conversion of data set names to lowercase, code page conversion, or adding a suffix to the filenames.	7. Copy from PDS(I)
Print a directory list to the ISPF list data set.	8. Print(P)
Compare two directories and put the results in an output file.	9. Compare(M)
Search all the files in a directory for a specified text string and put the results in an output file.	10. Find strings(F)
Change the working directory.	11. Set working directory(W)
Display the attributes for the file system the directory or a selected file is in.	12. File System (U)

Working with the Special File Pulldown

Using the **Special File** pulldown, you can perform the following tasks with a FIFO special file or a symbolic link.

Table 5. Using the ISPF Shell to Work with a FIFO Special File or a Symbolic Link

Task	Pulldown Choice
Create a FIFO special file.	1. New FIFO
Create a symbolic link or an external link.	2. New symbolic link(N)
Display or modify permission for a FIFO special file.	3. Attributes(A)
Show attributes for a FIFO special file (such as date created, link count, and size).	3. Attributes(A)
Display the attributes and contents of a symbolic link.	3. Attributes(A)
Delete a FIFO special file or symbolic link.	4. Delete(D)
Rename a FIFO special file or symbolic link.	5. Rename(R)
Create a hard link to a FIFO special file.	6. Link

Working with the Tools Pulldown

Using the **Tools** pulldown, you can perform the following tasks.

Table 6. Using the ISPF Shell's Tools Pulldown

Task	Pulldown Choice
Display process attributes or send a signal to a process	1. Work with processes(PS)
Run a shell command, using the login shell **/bin/sh -Lc**	2. Run shell command(SH)
Run a program or shell command, using the spawn service instead of a shell	3. Run program(EX)

Working with the File Systems Pulldown

Using the **File Systems** pulldown, you can perform the following tasks with a file system.

Table 7. Using the ISPF Shell to Work with a File System

Task	Pulldown Choice
Display all mounted file systems.	1. Mount table
Display the attributes of a mounted file system (such as total blocks, blocks in use, and ddname).	1. Mount table
Allocate an HFS data set.	2. New
Mount a File System (**superuser only**)	3. Mount (O)

Working with the Options Pulldown

Using the **Options** pulldown, you can select the following customization options.

Table 8. Tailoring the ISPF Shell for Your Use

Task	Pulldown Choice
Select sorting options for a directory list (for example, case-insensitive sort by filename).	1. Directory list

Table 8. Tailoring the ISPF Shell for Your Use (continued)

Task	Pulldown Choice
Select display options for a directory list (for example, show file type or permissions).	1. Directory list
Select the default action to be taken for each file type when you type a pathname on the main panel and press <Enter>.	2. Default actions
Select Edit and Browse options.	3. Edit or browse
Enter Edit profile name.	3. Edit or browse
Enter Edit initial macro and select option to bypass the initial macro panel when you select Edit.	3. Edit or browse
Position the command line on the bottom of the screen during an Edit or Browse session.	3. Edit or browse
Change the position of the command line.	4. Command line on top
Activate or deactivate the prompt that asks you to confirm a deletion (for example, deleting a file).	5. Bypass delete confirmation
Activate or deactivate the prompt that asks you to confirm that you want to exit the shell.	6. Bypass exit confirmation

While using the ISPF shell, you can enter ISPF system commands to perform customization tasks, such as:

- KEYLIST, to customize the function keys
- PFSHOW, to control the display of function keys

Enter the command from the command line in the ISPF shell. (You can also enter the LIST command to process the ISPF list data set.) For more details on these ISPF system commands, see *ISPF Dialog Management Guide and Reference*

System Programmer Tasks

A system programmer can use the ISPF shell to perform the following tasks, which are further discussed in *OS/390 UNIX System Services Planning*. These tasks require superuser authority.

Note: For installations with a large number of users, using the ISPF shell to authorize users to OMVS will eventually result in a storage shortage failure (reported as an ABEND66D RC40 in IKJEFT56). This failure occurs when the ISPF shell issues a LISTUSER / LU command to find the highest UID - in order to issue the next sequential number to the new OMVS user being defined. Due to the amount of output that can be generated by the LU command, a storage shortage below the 16MB line occurs. You should be aware of this limitation in the ISPF shell and use alternate methods to administer and maintain OMVS users.

Table 9. Using the ISPF Shell for System Programmer Tasks

Task	Action Bar Choice	Pulldown Choice
Mount a file system.	File systems	3. Mount(O)
Unmount a file system.	File systems	1. Mount table
Reset a pending unmount.	File systems	1. Mount table
Reset a quiesce status.	File systems	1. Mount table
Change attributes for an OS/390 UNIX user.	Setup	1. User

Table 9. Using the ISPF Shell for System Programmer Tasks (continued)

Task	Action Bar Choice	Pulldown Choice
Authorize an OS/390 UNIX user.	Setup	1. User
Display a list of users and sort by name, UID, or GID.	Setup	2. User list
Print a list of users.	Setup	2. User list
Set up all OS/390 UNIX users.	Setup	3. All users
Set up kernel groups.	Setup	4. All groups
Permit users to alter their own home directory and initial program.	Setup	5. Permit field access
Create character special files.	Setup	6. Character special
Set up the root file system.	Setup	7. Root
Switch to or from superuser status.	Setup	8. Enable superuser mode(SU)

Chapter 12. Performance: Running Executable Files

Note: Information in this chapter is directed towards users of the OS/390 shell. Most examples pertain to the OS/390 shell and not the tcsh shell.

A process is a collection of threads that execute within an address space, along with the required system resources. A user's login shell is one example of a process.

- The OMVS command creates two processes per login: a process to control the terminal and then a process for the login shell.
- **rlogin** and **telnet** logins each create two processes: one to control the socket connection to the user, another for the login shell.
- Communications Server logins require only one process per login. Consequently, there is no method for requesting a shared address space for the Communications Server login shell.

Most utilities invoked from the shell command line run in new processes that the shell creates.

There is a systemwide limit on:
- The number of OS/390 UNIX processes across the system
- The number of OS/390 UNIX processes per user

For a discussion of these limits, see *OS/390 UNIX System Services Planning*.

The shell and other OS/390 UNIX commands and daemons can assign multiple processes to the same MVS address space; this is called a *shared address space*. Using a shared address space offers these advantages:

- A new process in the same address space can be started faster than a new process in another address space.
- A new process in the same address space requires fewer system resources (storage, for example) than a new process in another address space.

For **rlogin**, the system administrator must update **/usr/sbin/inetd.conf** by adding **-m** to the rlogind entry to enabled shared address space. When **-m** is added, the socket connection process and the login shell process share the same address space.

For the OMVS command, use the SHAREAS keyword to enable shared address space. When the SHAREAS keyword is used, the login shell process is nested in the user's TSO address space. Any other login shells started with the OMVS OPEN subcommand are also nested in the user's TSO address space. (With NOSHAREAS, other login shells started with the OMVS OPEN subcommand will each consume another address space.)

To enable shared address space for the shell, issue the command

```
export _BPX_SHAREAS=YES
```

interactively or place it in your **$HOME/.profile**. Then all simple commands (any command run in the foreground and that is not in a pipeline) will run in processes nested in the shell's address space. If this variable is not set or is not set to the value YES, the shell creates all processes in separate address spaces. No matter how the shell is started (with or without shared address space enabled), you must set _BPX_SHAREAS=YES if processes started by the shell itself are to run in processes nested in the shell's address space.

User applications can use shared address spaces as well. See the description of the spawn() function and the BPX1SPN and BPX1ATX callable services for details.

Some processes cannot execute correctly in a shared address space. For example, if a process needs to reserve MVS system resources that are common to all processes in an MVS address space, it must run by itself. If two processes using the same MVS resource attempted to execute concurrently in the same address space, they would compete for these resources thus causing at least one of them to fail. When a potential storage shortage is detected, the new processes are created in their own address spaces, even if _BPX_SHAREAS=YES is present in the invoker's environment. For more details about these restrictions, see the descriptions of the spawn() function and BPX1SPN callable service.

Improving Shell Script Performance

You may be able to improve shell script performance by setting the **_BPX_SPAWN_SCRIPT** environment variable to a value of YES. However, when **_BPX_SPAWN_SCRIPT=YES**, the behavior will not conform completely to the XPG4 Commands & Utilities specification.

See "Improving the Performance of Shell Scripts" on page 44 for more information.

Improving Performance of the make Utility

It may be possible to improve the performance of **make**, by setting the shell variable as:

```
_MAKE_BI=YES
```

When this shell variable is set, sh will invoke the built-in versions of make, c89, c++, and cc instead of the /bin commands. Using the built-in versions may be beneficial when making large applications since the shell does not need to start another process. Instead, the shell calls the built-in **make**. As with other shell variables, this may be set in /etc/profile, $HOME/.profile or at the command line by the user.

For more information about built-in commands see *OS/390 UNIX System Services Command Reference*.

Chapter 13. Communicating with Other Users

You can communicate only with users in the same environment you are working in. For example, if you are working in the TSO/E environment, you cannot use MVS facilities to send a message to a user working in the shell.

Shell users who want to exchange messages with other shell users at the same system can use shell commands. Other users may prefer to use TSO/E facilities in order to be able to exchange messages with all TSO/E users, not just those using the shell.

Within the shell, you can send and receive messages using these shell commands:
- **mailx**
- **mail**
- **write**
- **talk**
- **wall**

Alternatively, you can switch into your TSO/E session and send messages to any TSO/E user by using TSO/E facilities, through either the OFFICE option of the Information Center Facility (ICF), if it is installed on your system, or through TSO/E commands. You can also receive messages using TSO/E.

If your system has Transmission Control Protocol/Internet Protocol (TCP/IP) or other network management facilities installed, you can log in to the TCP/IP network and send messages to users at other systems.

If your system has UUCP (Unix-to-Unix Copy Program) installed and set up, you can use this facility to send files to, or run commands or custom applications at, other sites in the UUCP network.

Using mailx to Send and Receive Mail

Using the **mailx** command, you can send a message. This command sends the message to a system-specified mail file. When the shell user receiving the message is ready to read messages, he or she uses **mailx** to see what messages have arrived and to read them.

Administrators and users can customize the behavior of **mailx** in a number of ways by selecting variables and setting them in files named **/etc/mailx.rc** and **$HOME/mail.rc**. Some variables apply for the duration of any session; you can set or reset others within a session.

The system programmer can set up a list of variables (using the **set** command) in the **/etc/mailx.rc** file. You can use these values as a default or you can set up a **$HOME/mail.rc** file that sets these variables for your personal use. These variables are described in *OS/390 UNIX System Services Command Reference* under the **mailx** command description.

You can reset certain variables during a session, and when entering **mailx** you can specify that the variables in the **/etc/mailx.rc** file are not to be used.

Sending Mail to Another User

You can send a message to one or more users at a time. The following example is a message sent to several users. The word in italics is output by **mailx** itself.

```
     mailx macneil
Subject: Reminder
     Our work group meets today at 10:30.
     Let's get together in the library.
     ~c smitha emilig fabish
     ~.
```

On the first line, the message is addressed just to macneil. The `~c` line adds people who will receive copies of the message.

The `~.` line identifies the end of the message and indicates to **mailx** that you are ready to send it. After you type that line and press <Enter>, the message is sent.

Here are the steps:

1. Type `mailx name`, where *name* is a login name.
2. The system prompts you for a `Subject`. You can type a word or phrase and press <Enter>.
3. Start typing the message. At the end of each line, press <Enter>. In the preceding example, you would press <Enter> after `Reminder`, `10:30.`, `library.`, and `fabish`.
4. To copy other people on the note, type `~c` before their login names.
5. To end the message and transmit it, type `~.` and press <Enter>. The system displays an `EOT` message.

Sending Mail to a Distribution List

You can send the same message to multiple users at the same time by using a distribution list.

If you use **mailx** to send a message, you can specify the *address* of each OS/390 UNIX user you want to receive the message. The simplest address is the TSO/E user ID. For example:

```
mailx pfeif lowell eliza fabish
```

requests that a message be sent to pfeif, lowell, eliza, and fabish. The shell then prompts for the subject and text of the message to be sent.

To send a message to a list of people, you can specify an *address alias* that contains a list of login names. For example, to set up an alias for the test team, you might enter the **mailx** subcommand:

```
alias test pfunt lulu detsch naga
```

After you do that, when you send a message to the address alias `test`, it will go to all the login names you specified on the **alias** command.

Aliases that are entered interactively remain in effect only for the current session. If you want to make the address alias permanent, put the **alias** command in your **.mailrc** startup file.

Sending a Message to an MVS Operator

You can use the **logger** shell command to send a message to an MVS operator, for example:

```
logger -d1 Is the tape I requested here yet?
```

This sends a message to a console with the route code 1.

Receiving Mail from Other Users

The simplest way to read incoming messages is to enter the command **mailx**. This starts an interactive session that lets you read your mail and perform other actions, such as display new messages, delete old ones, etc. If you do not have any mail, you will get a message telling you so.

When you have mail, the mail program shows you a list of messages similar to this one:

```
mailx xxxxxxx Type ? for Help.
"/usr/mail/SMITHA/...": 3 messages 3 new
>N  1 CLIFLWR           Thu Jul 15 14:28  6/93  testing
>N  2 HOMEBRW           Thu Jul 15 15:03  5/81  lunch plans
>N  3 ELVIS             Thu Jul 15 16:17  6/95  softball
?
```

The first line is the **mailx** program banner; xxxxxxx is information about the version of **mailx**. As indicated, you can type ? to see a help panel. The second line displays the name of the mailbox being used, /usr/mail/SMITHA/, followed by the number of messages in the mailbox, and their status. Then you see a list of three messages:

- Number 1 was sent by CLIFLWR and has the subject “testing”. It was sent on July 15 at 2:28 p.m., and contains 6 lines and 93 characters.
- Number 2 was sent by HOMEBRW and has the subject “lunch plans”. It was sent on July 15 at 3:03 p.m., and contains 5 lines and 81 characters.
- Number 3 was sent by ELVIS and has the subject “softball”. It was sent on July 15 at 4:17 p.m., and contains 6 lines and 95 characters.

The user names are all displayed uppercase.

The question mark (?) is the mail program prompt; it indicates that you can enter **mailx** subcommands now. Try the subcommand **n** (next message) to read the messages in sequence:

```
? n
Message  1:
From CLIFLWR Thu Jul  15 14:28
To: SMITHA
Subject:  testing

I'm setting up a meeting to test the toolkit
on Monday the 19th at 10AM.
Let me know if you can make it.
?
```

The question mark (?) prompt appears after the displayed message. You can also enter the **n** subcommand with a number to specify a particular message; for example, **n 3** displays the message about softball. Now you can choose what to do with the message: reply to it, save it, or delete it.

Replying to Mail

At the question mark (?) prompt, you can use the **R** (reply to sender) subcommand to reply to a particular message. This is an uppercase **R**: it differs from the **r**

subcommand, which sends the reply to everyone who sent and received the message. When you give the **R** subcommand, follow it with the message number. For example:

```
? R 1
To:  cliflwr
Subject:  Re:  testing

Yes, I can make the meeting.  where ?
~.
EOT
```

The EOT indicates that your reply has been sent.

Saving and Deleting Mail

If you exit mail without specifically deleting or saving your messages, the system saves those messages.

To save a message, use the **s** subcommand and give the name of the file you want to save the message in; for example:

```
s climail
```

If this is an existing file, the message is appended to it. If the file does not exist, it is created.

To delete a message, use the **d** subcommand and give the number of the message you want to delete:

```
? d 1
?
```

The mail program deletes message number 1 and returns another ? prompt.

Ending the mailx Program

To exit from **mailx**, use the **q** (quit) subcommand:

```
? q
$
```

The shell prompt indicates that you have left mail and can enter shell commands again.

For more information on **mailx**, see *OS/390 UNIX System Services Command Reference*.

Using write to Send a Message or a File

The **write** command lets you send a message directly to someone else who is logged on to the system. To determine who is logged on, use the **who** command. The **who** command displays information about who is logged on in this form:

```
BUBBA    ttyp0002                Feb 8 09:49
```

where `BUBBA` is a login name, `ttyp0002` is the terminal, and `Feb 8 09:49` is the login time.

The typical format of the **write** command is:

`write` *`user_name`*

However, if a user is logged in more than once, you can specify *terminal* (in the `ttyp` form that **who** returns) rather than *user_name*.

Sending a Message: An Example

Here is an example of how to send a message, using **max** as the sender and **bubba** as the recipient:

```
write bubba
```

When **max** sends a message to **bubba**, **bubba** receives a message like this:

```
Message from max (ttyp002) [Feb  8 15:04 ] ...
```

After the system establishes the connection to **bubba**, it sends two alert characters (usually a beeping sound) to **max**'s terminal to indicate that it is ready to send a message. **max** can then type a message, which appears on **bubba**'s terminal. If a message is more than one line, each time you press <Enter> a line is sent to **bubba**'s terminal.

Ending a Message

To end a message, enter <EscChar-D> for end-of-file or <EscChar-C> for an interrupt. When **write** receives an end-of-message indicator, it displays an `EOF` message on the other user's screen and breaks the connection.

When your message is completed, the other user can reply to your message with

```
write your_user_name
```

However, if both of you are trying to write on each other's terminal at the same time, the messages may get interleaved on your screens, making them difficult to read. For two-way conversations, use **talk** instead of **write**. For more information about **talk**, see "Using talk for an Online Conversation".

Sending a File

You can add the output of a command to a message that you are writing. To do this, start a line with an exclamation mark (`!`) and put a standard shell command on the rest of that line. **write** calls your shell to execute the command, and sends the standard output (**stdout**) from the command to the other user. The other user does not see the command itself or any input to the command. For example, you might write:

```
Here is what my file contains:
!cat file1
```

The contents of **file1** are displayed on the other user's screen.

Using talk for an Online Conversation

talk lets you start up a two-way conversation with someone else logged in to the system. However, **talk** is available only if you access the shell with **rlogin** or **telnet** or the Communications Server, because it requires raw mode.

The typical format of the **talk** command is:

```
talk user_name
```

However, if a user is logged in more than once, you can specify *terminal* (in the form `ttyp` that **who** returns) rather than *user_name*.

Beginning a Conversation: An Example

Here is an example of how to begin a conversation with **talk**, using **max** as the person starting a conversation with **bubba**. Here **max** begins by typing:

```
talk bubba
```

bubba receives a message like this:

```
Message from max.
talk: connection requested by max
talk: respond with: talk max
```

To set up the two-way connection, **bubba** must enter:

```
talk max
```

After this connection has been established, the two can type simultaneously.

Viewing the Conversation

talk displays incoming messages from the other person in one part of the screen and your outgoing messages in another part of the screen.

Some terminals may not be able to split the screen into parts in this way. Depending on the terminal type, **talk** may try to simulate this effect. However, it may not be possible for both users to enter messages simultaneously.

Using wall to Broadcast Messages

A superuser can use the **wall** command to send a message to all logged in shell users:

wall [*message*]

If the message is omitted from the command line, the user will receive two beeps as a prompt to enter the message. You input the message, pressing enter after each line, and when done inputting the entire message, enter end-of-file or an interrupt (typically, <EscChar-D> for end-of-file or <EscChar-C> for an interrupt).

The user of **wall** should be a superuser. This ensures that the user is permitted to write to all the users that are logged on. If a user who is not a superuser attempts to use **wall** to broadcast a message, some writes will fail and those users will not receive the message.

Users who are sent a broadcast message will receive a beep announcing the message, and a message in the form:

```
Broadcast Message from SWER@AQFT (ttyp0006) at 10:43:54 (EDT5EST) ...

This is the text of the message line1
This is line2
```

For more information on the **wall** command, see *OS/390 UNIX System Services Command Reference* .

Controlling Messages and Online Conversations

You can use the **mesg** command to control whether other users can send messages to your terminal with **talk**, **write**, or similar commands.

To let other people send you messages, issue:

```
mesg y
```

To tell the system not to let other people send you messages, issue:

```
mesg n
```

To display the current setting without changing it, issue:

```
mesg
```

Using the UUCP Network

If your system administrator has UUCP (Unix-to-Unix Copy Program) set up to communicate with remote sites, you can use this facility to send or retrieve files, or to run commands or custom applications at other sites in the UUCP network. To send or retrieve files from remote sites, use the **uucp** command; this causes a file transfer request to be queued. Depending on how your system is set up, a file transfer request may be processed immediately or later at a scheduled time.

UUCP provides the **uucp** command, which schedules files to be exchanged with other UUCP systems, and the **uux** command, which schedules commands to be executed by other UUCP systems. However, the **uucp** and **uux** commands do not cause any files to be exchanged or commands to be executed. For this, UUCP provides two daemons called **uucico** and **uuxqt** which establish communication sessions, transfer data, and execute commands according to the requests scheduled by **uucp** and **uux**.

The commands that you will use with UUCP are:

uucp Copy files between remote systems

uuname
Display a list of UUCP systems

uupick
Manage files sent to you via **uuto**

uustat Display the status of pending UUCP transfers

uuto Copy files to users on remote systems

uux Request command execution on remote systems

Code Page Conversion: **uucp**, **uuto**, and **uupick** do not convert file data to or from EBCDIC. The sending and/or receiving user must convert file data if two systems have different codesets. You can use the **iconv** command to do this.

Transferring a File to a Remote Site

To transfer a file to a remote site, use the **uucp** command or the **uuto** command.

Using uucp to Transfer Files

uucp automatically handles text and binary files. When a file is transferred by **uucp** to another site, it is put in the public UUCP directory—by default, this is **/usr/spool/uucppublic**.

1. You need to know the name of the remote site. To list the remote sites that have been configured, type:

```
uuname
```

The sites are listed, one per line.

2. Copy the file to the other site.

   To make file transfers easier, you can use a special character in pathnames for the public UUCP directory. When tilde (~) is written as the first directory in a destination path name, the **~/** stands for the public UUCP directory. You can specify the public UUCP directory with the pathname **~/**.

   For example, to copy the file `memo1.pay` in your current directory to the public directory on the site named `north`, type:

   ```
   uucp memo1.pay north!~/memo1.pay
   ```

   File transfers may not get processed immediately. If there is any chance that the file to be sent will not be available later, use the **-C** option on the **uucp** command to immediately copy the file to the **uucp** spool directory. This ensures that the file is available later when the file transfer occurs.

Using uuto to Transfer Files

uuto is a simplified method of invoking **uucp**, and it also handles text and binary files automatically. When a file is transferred by **uuto** to another site, it is put in the **receive/***usr* subdirectory of the public UUCP directory. Within the **receive** subdirectory, each user on the local system has a subdirectory. For example, a file for user stiert would be transferred to **/usr/spool/uucppublic/receive/stiert**.

1. You need to know the name of the remote site. To list the remote sites that have been configured, type:

   ```
   uuname
   ```

   The sites are listed, one per line.

2. Copy the file to the other site. For example, to copy the file `memo1.pay` in your current directory to the public directory on the site named `north`, type:

   ```
   uuto memo1.pay north!nuucp
   ```

   The recipient is notified by mail when the file arrives. To get the file, the recipient should use the **uupick** command. See “Working with Your Files in the Public Directory” on page 188 for information on how to use the **uupick** command.

Transferring Multiple Files to a Remote Site

You can use **uucp** to transfer more than one file, specifying the files by name or by using wildcards. To send more than one file, you must specify a directory as the destination, not a file name. To do this, end the destination pathname with a slash (/).

For example, to send the files **jan.wks**, **feb.wks**, **mar.wks**, and **memo1.txt**, to the directory **receive** at the `north` site, type:

```
uucp *.wks memo1.txt north!~/receive/
```

The trailing slash (/) shows that **receive** is a directory.

You can send an entire directory, by specifying the contents of the directory with a wildcard.

Transferring a File to the Local Public Directory

You may want to put a file in your local public directory so that others can access it there. To specify the public directory in a local pathname, put single quotes around the pathname so that the shell does not treat the tilde as your home directory. (For more information on how the shell interprets a tilde in file names, see “Characters Used in Filenames” on page 78.)

For example, to copy that file to your own UUCP public directory, type:

```
uucp memo1.pay '~/memo1.pay'
```

Notification of Transfer

If you want to be certain that a file has been transferred or if you want someone at the remote site to know that the file has arrived, you can use the *-m* and *-n* options on the **uucp** command, or the *-m* option on the **uuto** command.

- With **uucp -m** or **uuto -m**, as soon as the file is successfully transferred, you receive a mail message. You can use **mailx** to read the message. The first line describes the file transfer request, and the second line describes the result. For example, it might look like this:

  ```
  REQUEST: home!/usr/spool/uucppublic/memo1.txt -->
      north!/usr/spool/uucppublic/memo1.txt
  (SYSTEM north) copy successful
  ```

- With **uucp -n** *name*, if you are transferring a file to a remote site, you can specify the login name of the person at the remote site to be notified when the file is transferred. That person can read the notification message using **mailx**.

Permissions

Each site in a UUCP network has a **Permissions** file that is used to control the access that remote systems have to data and programs on the local system. This file is used to specify, among other options, the areas in the file system that a remote system can read or write from, the commands that the remote system can run on the local system, and a different public directory than the default. Those options are specified as:

READ Indicates which directories can be read. By default, this is the home directory of user uucp (**/usr/spool/uucppublic**).

WRITE
Indicates which directories **uucico** can write to. By default, this is **/usr/spool/uucppublic**, the home directory of user uucp.

NOREAD
Indicates that files in the specified directories cannot be read. If a directory is specified by both `READ` and `NOREAD`, files in that directory cannot be read. The public directory can always be read (even if specified on `NOREAD`).

NOWRITE
Indicates that files in the specified directories cannot be written to. If a directory is specified by both `WRITE` and `NOWRITE`, files in that directory cannot be written to. The public directory can always be written to (even if specified on `NOWRITE`).

PUBDIR
Indicates the public directory. By default, this is the home directory of user uucp (**/usr/spool/uucppublic**).

COMMANDS
Indicates the commands that the remote system can execute on your

system. If more than one command is specified, the command names are separated with a colon (:). For example, `COMMANDS=uucp:ls`. If all commands are prohibited, the `COMMANDS` option is not used.

For a full description of all the Permissions file options, see *OS/390 UNIX System Services Planning*.

Transferring a File from a Remote Site

To copy a file from a remote site, your site must have read permissions on the file. Normally your site would have read permissions only on the public UUCP directory and its subdirectories.

For example, say you want to copy the program **pages** from **programs**, a subdirectory of the remote site's public UUCP directory, to your public UUCP directory.

To retrieve the file, you would enter this command:

```
uucp south!~/programs/pages '~/pages'
```

where `south` is the remote site.

For more information about the **uucp** command, see *OS/390 UNIX System Services Command Reference*.

Checking a File's Transfer Status

To check the status of pending transfer requests, use the **uustat** command. You can specify options to display the status of transfers for a particular job ID or user ID.

To display completed file transfer attempts, use the **uulog** command. To see the record of completed file transfer attempts and connections by site, type:

```
uulog -s site
```

where *site* is the name of the remote site.

For more information about the **uustat** and **uulog** commands, see *OS/390 UNIX System Services Command Reference*.

Working with Your Files in the Public Directory

All users have read access to the UUCP public directory. When you have a file in the public directory, you can use the **cp** command to copy the file or the **mv** command to move the file. If the sender uses the **-n** option on **uucp**, you are notified when the file is placed in the public directory.

Files sent to you with the **uuto** command are automatically placed in the **receive** subdirectory. You can use **uupick** to manage files in the **receive** subdirectory of the UUCP public directory. If **receive** is specified as the target directory on the **uucp** command, you can use **uupick** to manage the files.

Within the **receive** subdirectory, each user on the local system has a subdirectory.

To check your public UUCP directory for files sent to you by the **uuto** command, type **uupick**. For each file or directory found, **uupick** prompts you with a message and then you specify how that entry should be handled. For example, for a file, it might display:

```
from south:  file memo2.txt ?
```

In response, you could type **d** to delete the file, or **m** to move the file into your current working directory, or **m /mydir/tmp** to put it in the directory **/mydir/tmp**.

For more information about the **uupick** command, see *OS/390 UNIX System Services Command Reference*.

Running a Command on a Remote Site

You can use the **uux** command to run commands on remote sites, but they cannot be interactive commands such as **vi**. You must have a working UUCP connection and permission to execute commands on the remote site.

Using a Remote File as an Argument

To ask `south` to print the file **south!/schedule/january** using the **lp** command, you would type:

```
uux 'south!lp' '/schedule/january'
```

where **/schedule/january** is the name of the file on `south` to be printed. In general, if no site is specified on the arguments for the remote command, **uux** assumes the command is on the site running the command. You must specify full pathnames for files in **uux** commands. As a general rule, enclose all arguments to **uux** in single quotes to prevent the shell from interpreting them.

Using a Local File as an Argument

To ask `south` to print the local file **/schedule/january** using the **lp** command, you would type:

```
uux south!lp !/schedule/january
```

uux sends a copy of the file for printing; after the remote command has run, the copy is removed.

For more information about the **uux** command, see *OS/390 UNIX System Services Command Reference*.

Using TSO/E to Send or Receive Mail

You can use the TSO/E panel facilities or TRANSMIT and RECEIVE commands to communicate with any TSO/E user (including OS/390 UNIX users). If you use TSO/E to send a message, your correspondent must use TSO/E to receive it.

Sending a Message

You can use the TSO/E Information Center Facility (ICF), if installed, or TSO/E commands to send a message. For example, to send a short message (with no more than 115 characters), you can switch to TSO/E command mode and enter:

```
SEND 'Have to go home to take my cat to the vet' USER(alice)
```

You use SEND for messages to people on the same system as you.

For a longer message, or a message to someone on a different system, you could use:

```
TRANSMIT dallas.alice
```

where `dallas.alice` identifies the person to receive the message: `dallas` is the ID of the MVS system (known as a *node* in the network) where the person works, and `alice` is the person's user ID. The system then prompts you to enter the message.

Sending a Message to a Distribution List

You can use the TSO/E ICF, if installed, or the TSO/E TRANSMIT command to send a message to a distribution list. You set up a distribution list by specifying a nickname entry in the NAMES data set that contains a list of names or nicknames you want the message sent to.

For example, if you have set up the nickname `test` for a distribution list, you would type:

```
transmit test
```

The system displays a screen for input. You type your message and press <F3> to send it.

Sending a Message to an MVS Operator

To send a message to a specific MVS operator, you must know the operator's route code and specify it in the OPERATOR operand. For example:

```
SEND 'Are the tapes I wanted from the library here yet?' OPERATOR(7)
```

You can also send a message to a specific operator console by using the CN operand. A console name or ID is defined at your enterprise. Say you want to send this message to the operator console named TAPELIB: `Please send the tapes to the floor.` You would enter:

```
send 'please send the tapes to the floor.' CN(TAPELIB)
```

Receiving Mail from Other Users

How and whether you are notified when TSO/E messages are received by the system depends on how your TSO/E system is set up:

- You may be notified when you log on or as messages arrive.
- You may have to enter a RECEIVE command periodically to see if a message has arrived.

Unless the messages are automatically displayed when you log on, you enter a RECEIVE command to see your currently unread messages. For more information on TSO/E mail and messaging, see *OS/390 TSO/E User's Guide*.

Receiving Messages from Other Systems

TSO/E users can receive messages from other systems through the TSO/E message interface. Receiving a message from a user on another system is the same as receiving one from a user on the same system.

Part 2. The File System

Chapter 14. An Introduction to the Hierarchical File System

OS/390 UNIX files are organized in a hierarchy, as in a UNIX system. All files are members of a *directory,* and each directory is in turn a member of another directory at a higher level in the hierarchy. The highest level of the hierarchy is the *root directory*.

MVS views an entire file hierarchy as a collection of *hierarchical file system data sets (HFS data sets)*. Each HFS data set is a mountable file system. DFSMS/MVS facilities are used to manage an HFS data set, and DFSMS Hierarchical Storage Manager (DFSMShsm*)is used to back up and restore an HFS data set.

The root file system is the first file system mounted. Subsequent file systems can be mounted on any directory within the root file system or on a directory within any mounted file system.

A file in the hierarchical file system is called an *HFS file*.HFS files are byte-oriented, rather than record-oriented, as are MVS data sets. You can copy HFS files into MVS data sets (sequential data set, partitioned data set, or PDSE), and you can copy MVS sequential data sets or partitioned data set members into a hierarchical file system. However, it must be noted that you cannot share HFS files between LPARS.

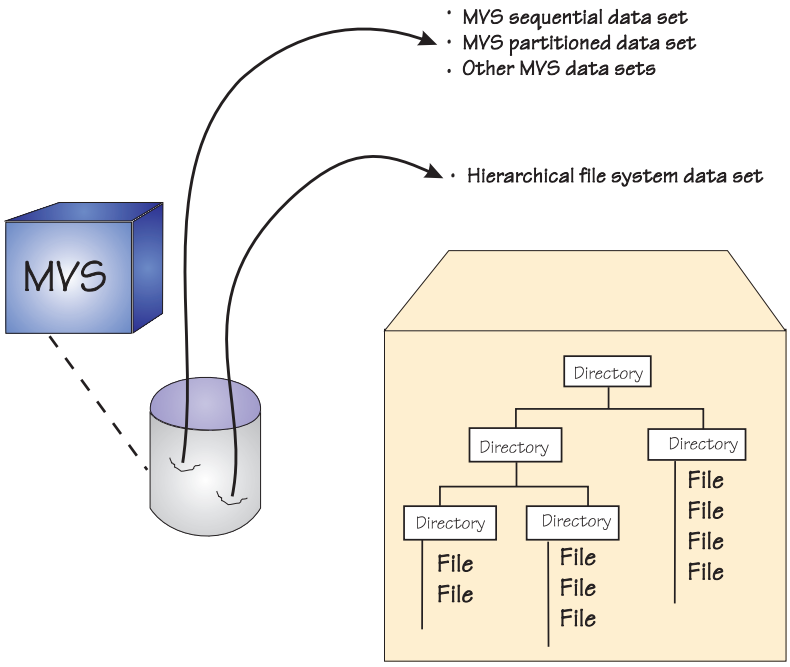


Figure 21. The Hierarchical File System

The OS/390 Shells and Utilities typically impose a *line orientation* on the byte-oriented files. A *line* is a stream of bytes terminated with a <newline> character.A line terminated by a <newline> character is sometimes referred to as a record. So, there is a single <newline> character between every pair of adjacent records. Text files use the <newline> character to delimit lines; binary files do not.

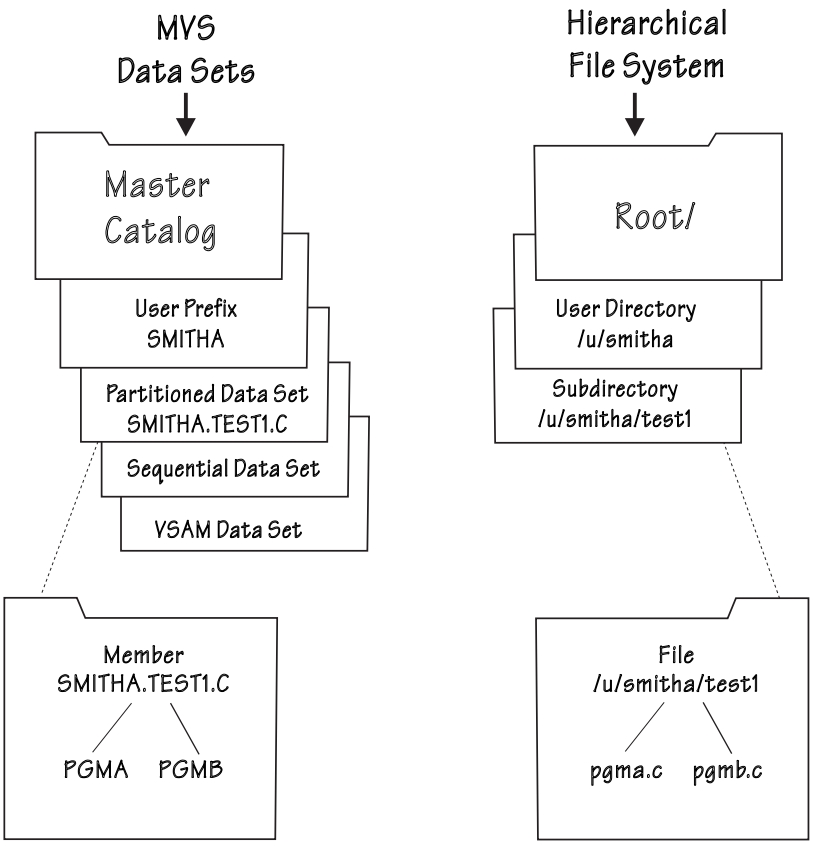


Figure 22. Comparison of MVS Data Sets and a Hierarchical File System

In Figure 22, you see that:

- The MVS master catalog is analogous to the root directory in a hierarchical file system.
- The user prefix assigned to MVS data sets is an organizer analogous to a user directory (**/u/smitha**) in the file system. Typically, one user owns all the data sets whose names begin with his user prefix. For example, the data sets belonging to the TSO/E user ID SMITHA all begin with the prefix SMITHA. There could be data sets named SMITHA.TEST1.C, SMITHA.TEST2.C, SMITHA.TEST1.LIST, and SMITHA.TEST2.LIST.

  In the file system, SMITHA would have a user directory named **/u/smitha**; under that directory there could be subdirectories named **/u/smitha/test1** and **/u/smitha/test2**.
- Of the various types of MVS data sets, a partitioned data set (PDS) is most akin to a user directory in the file system. In a partitioned data set such as SMITHA.TEST1.C, you could have members PGMA, PGMB, and so on—for example, SMITHA.TEST1.C(PGMA) and SMITHA.TEST1.C(PGMB). Likewise, a subdirectory such as **/u/smitha/test1** can hold many files, such as **pgma.c**, **pgmb.c**, and so on.

All data written to the hierarchical file system can be read by all programs as soon as it is written. Data is written to a disk when a program issues an **fsync()**.

Before learning about the file system capabilities, you need to understand these concepts:

- The root file system and mountable file systems

- Directories
- Files
- Path and pathname

The Root File System and Mountable File Systems

Taken as a whole, the *file system* is the entire set of directories and files, consisting of all HFS files shipped with the product and all those created by the system programmer and users. The system programmer (superuser) defines the *root file system*; subsequently, a superuser can mount other *mountable file systems* on directories within the file hierarchy. Altogether, the root file system and mountable file systems comprise the file hierarchy used by shell users and applications.

After installation of OS/390, the end user's logical view of the file system is as shown in Figure 23.

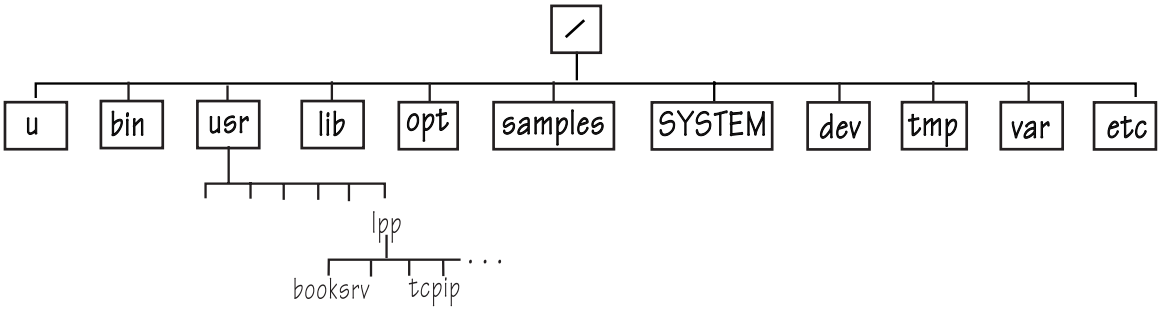


Figure 23. End User's Logical View of the File System

System programmers need to know that the illustration of directories in Figure 23 is not a true representation of file systems. Starting in R9, the HFS, as installed through either ServerPac or CBPDO, consists of **/dev, /tmp, /var,** and **/etc** symbolic links that point to the **/dev, /tmp, /var,** and **/etc** directories, as demonstrated in Figure 24.

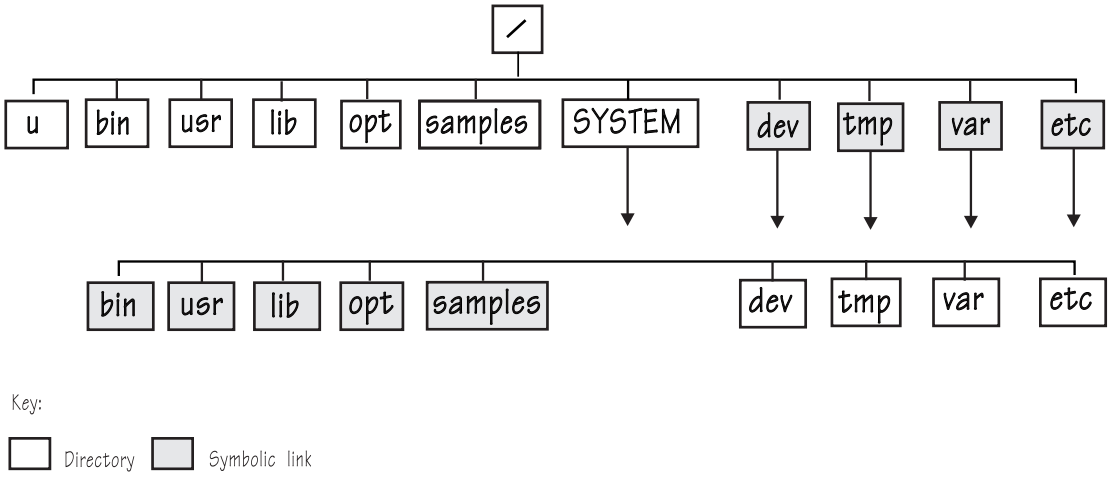


Figure 24. Organization of the File System in R9

Here SYSTEM is an HFS data set that contains directories which are used as mount points, specifically for the **/etc, /var, /tmp,** and **/dev** file systems. Also starting in R9, IBM requires that you mount **/etc, /var, /dev,** and **/tmp** in separate HFS data sets.

Several types of file systems can be mounted within the file hierarchy:

- Hierarchical File System (HFS)
- Network File System (NFS): Using NFS client on OS/390 UNIX System Services, you can mount a file system, directory, or file from any system with an NFS server within your user directory. You can edit or browse the files.
- Distributed File System (DFS): A DCE component, DFS joins the local file systems of several file server machines making the files equally available to all DFS client machines. DFS allows users to access and share files stored on a file server anywhere in the network, without having to consider the physical location of the file.
- Temporary File System (TFS): The TFS is an in-memory physicalfile system that delivers high-speed I/O. To take advantage of that, the system programmer (superuser) can mount a TFS over the **/tmp** directory so it can be used as a high-speed file system for temporary files. (Normally, the TFS is the file system that is mounted instead of the HFS if the OS/390 UNIX system services are started in minimum setup mode.)

A directory can include a file that is itself a directory (sometimes referred to as a *subdirectory*) and so on, through a number of levels in a hierarchical arrangement. For example, in Figure 23 on page 195, the slash (/) symbol at the top represents the *root directory,* which all other directories are descended from. There are ten directories branching from the root. Each of these directories, in turn, has its own system of subdirectories and files. For example, **localedef** is a subdirectory in the directory **/usr/lib/nls**.

In the file system, such I/O destinations as terminals, data buffers, and queues (such as named pipes) are defined as files. For example, a workstation could have the filename **/dev/ttyp0001**.

Finding the HFS Data Set that Contains a File

To determine which HFS data set (filesystem) contains a file, use the **df** shell command. You can use **df** to find the HFS data set (filesystem) that contains your current working directory or **df** *filename* to find the HFS data set of another file.

Directories

Files are grouped in a *directory,* which is a special kind of file consisting of the names of a set of files and other information about them. Usually, the files in a directory are related to each other in some way. The files listed can be thought of as being “kept” in that directory (although their actual locations in physical storage are managed by the operating system).

A directory can have subdirectories. For example, in a user’s directory **/projectx**, there might be subdirectories such as **/projectx/design** and **/projectx/test**.

When you first enter the OS/390 shell, you are automatically placed in your *home directory,* which is defined when your user ID is defined.

Files

There are four other types of files that can exist in the HFS, in addition to directories:

- A **regular file**is an identifiable (named) unit of text or binary data information. A file can be C source code, a list of names or places, a printer-formatted document, a string of numbers organized in a certain way, an employee record containing smaller information units in fields, a memo, and many other possible things. A user or an application program must understand how to access and use the individual increments of information (such as employee record fields) within a file.
- A **character special file** defines one of these:
  - A terminal (**/dev/ptyp***nnnn* and **/dev/ttyp***nnnn*). Only a superuser can create this file.
  - The default controlling terminal for a process (**/dev/tty**).
  - A null file (**/dev/null**). Data written to this file is discarded; hence, it is known as “the bit bucket”. Only a superuser can create this file.
  - A file descriptor file (**/dev/fd***n*). Only a superuser can create this file.
  - A system console file (**/dev/console**). Data written to this file is sent to the console using a write-to-operator (WTO) that displays the data on the system console. Only a superuser can create this file.
  - A UNIX domain socket name file. This is a pathname that specifies the socket address for a UNIX domainsocket. The pathname is assigned by the application programmer; there is no convention for the name. The operating system creates the file.
  - A Communications Server remote tty file (for example, **rtynnnn**) that corresponds to the requesting terminal on the originating Communications Server node. The name is assigned by the Communications Server administrator.
  - The Communications Server character special file (**/dev/ocsadmin**) that supports **ioctl** functions for Communications Server administrative functions.
- A **FIFO special file** is a file typically used to send data from one process toanother so that the receiving process reads the data first-in-first-out (FIFO). A FIFO special file is also known as a *named pipe*.
- A **symbolic link**is a file that contains the pathname for another file, in essence a reference to the original file. Only the original pathname is the real name of the original file. You can create a symbolic link to a file or a directory. In R9 and later, **/etc, /tmp, /dev,** and **/var** are symbolic links.

  An *external link* is a type of symbolic link,a link to an object outside of the HFS. Typically, it contains the name of an MVS data set.

Users and programs create regular files, FIFO special files, symbolic links, and external links.

Files Not in the HFS: There are two types of unnamed files that you may be aware of, but they do not exist in the HFS:

- An **unnamed pipe.**

  A program creates a pipe with the **pipe()** function. A pipe typically sends data from one process to another; the two ends of a pipe can be used in a single program task. A pipe does not have a name in the file system, and it vanishes when the last process using it closes it.
- A **socket**.

A program creates a socket with the **socket()** function. A socket is a method of communication between two processes that allows communication in two directions, in contrast to pipes, which allow communication in only one direction. The processes using a socket can be on the same system or on different systems in the same network.

Executable Modules in the File System

You can have an executable module in HFS. To run a shell script or executable, a user must have read and execute permissions to the file. Use **chmod** to set the permissions.

- For frequently used programs in the HFS, you can use the **chmod** command to set the *sticky bit*. This reduces I/O and improves performance. When the bit is set on, OpenEdition MVS searches for the program in the user's STEPLIB, the link pack area, or the link list concatenation. For further information, see "Using a Symbolic Mode to Specify Permissions" on page 237.
- The **extattr** command is used to set, reset and display extended attributes for files to allow executable files to be marked so they run APF authorized, as a program controlled executable, or not in a shared address space.

  The **ls** shell command has an option which will display these attributes:

  -E Displays extended attributes for regular files:

  a Program runs APF authorized if linked AC=1

  p Program is considered program controlled

  s Program runs in a shared address space

  - Attribute not set

  When the **extattr** attribute **l** is set (**+l**) on an executable program file, it will be loaded from the shared library region.
- You can copy executable modules between MVS and HFS. For more information on how to do this, see "Copying Executable Modules between MVS and the File System" on page 293.
- For information on how to set up a STEPLIB environment for an executable file, see "Building a STEPLIB Environment: The STEPLIB Environment Variable" on page 50.

For more information on the **ls** and **extattr** shell commands, see *OS/390 UNIX System Services Command Reference*.

Path and Pathname

The set of names required to specify a particular file in a hierarchy of directories is called the *path* to the file, which you specify as a *pathname*. Pathnames are used as arguments for commands.

An *absolute pathname* is a sequence that begins with a slash for the root, followed by one or more directory names separated with slashes, and ends with a directory name or a filename. The search for the file begins at the root and continues through the elements in the pathname until it gets to the final name. For example:

```
/u/smitha/projectb/plans/1dft
```

is the absolute pathname for **1dft**, the first draft of the plans for a particular project that a user named Alice Smith (SMITHA) is working on.

Instead of using the absolute pathname with shell commands, you can specify a pathname as relative to the working directory; this is called the *relative pathname*. In most cases, a user can specify a particular file without having to use its absolute pathname. A relative pathname does not have a / at the beginning, and the search for the file begins in the working directory. For example, if Alice Smith is working in the directory **projectb**, she can specify the relative pathname for the file **/u/smitha/projectb/plans/1dft** as:

```
plans/1dft
```

A pathname can be up to 1023 characters long, including all directory names, filenames, and separating slashes. For pathnames and filenames, use characters from the POSIX portable character set. Using DBCS data in these names is not recommended; it may cause unpredictable results.

The system performs *pathname resolution* to resolve a pathname to a particular file in a file hierarchy. The system searches from element to element in a pathname in order to find the file.

Requirement for an Absolute Pathname

In some situations, an absolute pathname is required. Table 10 shows that job control language (JCL) and some TSO/E commands require an absolute pathname and that they require an MVS data set name to be specified in a certain way. In these situations, the maximum length of the absolute pathname is 255 characters.

Table 10. Absolute Pathname Requirements

	Pathname	**Dataset name**
JCL	Absolute, in single quotes	Fully qualified (no quotes needed).
ALLOCATE command	Absolute, in single quotes	Fully qualified in single quotes. If specified without quotes, the TSO/E prefix is added to the data set name. Normally the TSO/E prefix is the TSO/E user ID (this can be changed with the PROFILE PREFIX( ) command).
OEDIT and OBROWSE commands	Absolute, unless you are working in your home directory	Not Applicable
OPUT, OGET commands	Absolute (unless you are working in your home directory), in single quotes	Fully qualified in single quotes. If specified without quotes, the TSO/E prefix is added to the data set name. Normally the TSO/E prefix is the TSO/E user ID (this can be changed with the PROFILE PREFIX( ) command).
OPUTX, OGETX commands	Absolute (unless you are working in your home directory)	Fully qualified in single quotes. If specified without quotes, the TSO/E prefix is added to the data set name. Normally the TSO/E prefix is the TSO/E user ID (this can be changed with the PROFILE PREFIX( ) command).

Resolving a Symbolic Link in a Pathname

A symbolic link is a file that contains the pathname for another file; that pathname can be relative or absolute. If a symbolic link contains a relative pathname, the pathname is relative to the directory containing the symbolic link.

If you use a symbolic link as a component of a pathname, during pathname resolution the original pathname is changed. How it changes depends on whether the symbolic link contains a relative or absolute pathname. For example, consider the pathname **/u/turbo/dlg/lev1**:

- If **dlg** is a symbolic link containing the relative pathname **dbopt/pgma/src**, **dlg** is replaced by the relative pathname. This is how it resolves:

  /u/turbo/dlg/lev1 → **/u/turbo/dbopt/pgma/src/lev1**.
- If **dlg** is a symbolic link containing the absolute pathname **/usr/bin/dbopt/pgma/src**, then the components in the original pathname that preceded **dlg** are replaced by the absolute pathname in the symbolic link. This is how it resolves:

  /u/turbo/dlg/lev1 → **/usr/bin/dbopt/pgma/src/lev1**.

Up to eight symbolic links can be resolved in a pathname.

Note: An external link is a type of symbolic link that refers to an object outside of the hierarchical file system. As used by the Network File System feature, an external link refers to an MVS data set name.

Symbolic and External Links with a Sticky Bit

Note: DLLs, and all flavors of **spawn()** and **exec()**, follow the same processing as described below. Where it says **exec()**, it covers all forms of module loading.

1. External links:

   exec() does a **stat()** on the passed filename. **stat()** does the search, not **exec()**. If the filename is an external link, then **stat()** fails with a unique reason code which causes **exec()** to read the external link. If the external link name is a valid PDS member name (1–8 alphanumeric/special characters), then **exec()** will attempt to locate the module in the MVS search order. If it cannot be found, **exec()** fails.

   The external link is normally used when you want to set the sticky bit on for a file name which is longer than 8 characters or contains characters unacceptable for a PDS member name.
2. Symbolic links:

   If the filename you specify is a symbolic link, and **exec()** sees the sticky bit on, then it will truncate any dot qualifiers. So, as long as the base filename is an acceptable PDS member name, the need to set up links in order to get **exec()** to go to the MVS search order should not be an issue.

   For example, if you have a file named java.jll, when you put the sticky bit on, **exec()** will attempt to load JAVA. If **exec()** cannot find JAVA, it will revert to using the java.jll file in the file system.

   The important thing to understand is that **exec()** never sees the name that the symbolic link resolves to, even though it can see the **stat()** data for the final file.

If you define **/u/user1/name1** as a symbolic link to **/u/user1/name2**, and then invoke *name1*:

1. The shell will spawn *name1*.
2. **spawn()** will access the file for *name1* unaware that there is a symbolic link already established. It will access the *name2* file by its underlying vnode, not the *name2* handle.
3. If the sticky bit is on for the *name2* file, **spawn()** will do the MVS search for *name1* (the only name it has to work with).

Command Differences Due to Symbolic Links

In Release 9 and later, certain directories like **/etc, /dev, /tmp,** and **/var** are converted to symbolic links. Some shell commands have minor technical differences when referring to symbolic links than for regular files or directories. For example, **ls** does not follow symbolic links by default. Prior to Release 9, **/etc** was a directory, so `ls /etc` would display all files in **/etc**. In Release 9, **/etc** is a symbolic link, so `ls /etc` will display only the symbolic link name, in this case **/etc**.

In order to follow symbolic links, you must specify `ls -L` or provide a trailing slash. For example, `ls -L /etc` and `ls /etc/` both display the files in the directory that the **/etc** symbolic link points to.

Other shell commands that have differences due to symbolic links are **du, find, pax, rm** and **tar**.

While these behavioral changes should be minor, users can tailor command defaults by creating aliases for the shell command. For example, if you want **ls** to follow symbolic links, you could issue the command `alias ls="ls -L"`. Aliases are typically defined in the users' **ENV** file. Refer to *OS/390 UNIX System Services Command Reference* for details on the **alias** command.

Note: After this alias has been established, then **ls** will follow all symbolic links.

An administrator can put **alias** commands in **/etc/profile** which could affect all users' login shells. IBM does not recommend this, because changing the default behavior in **/etc/profile** may produce unexpected results in shell scripts or by shell users.

Using Commands to Work with Directories and Files

There are numerous shell commands you can use to create and work with directories and files. See "Managing Files" on page 307 for a list of them.

To get online help for using the shell commands, you can use the OHELP command or the **man** command. See "Online Help" on page 89 for more information.

You can also use TSO/E commands to do certain tasks with the file system. Some of these are tasks that UNIX users traditionally perform while in the shell.

Command	Task
ISHELL	Invoke the ISPF shell. This is a panel interface for performing many user and administrator tasks. For more information, see "Using the ISPF Shell" on page 168.
MKDIR	Create a directory. Unlike the **mkdir** shell command, this command does not create intermediate directories in a pathname if they do not exist.
MKNOD	Make a character special file. To use this command, you must be a superuser.
MOUNT	Add a mountable file system to the file hierarchy. To use this command, you must be a superuser.

OBROWSE	Browse (read but not update) an HFS file using the ISPF full-screen browse facility.
OCOPY	Copy an MVS data set member into an HFS file, using ddnames. Copy an HFS file into an MVS data set member, using ddnames. Copy an HFS file into another HFS file. Copy an MVS data set member into another MVS data set member.
OEDIT	Create or edit text using the ISPF editor.
OGET	Copy an HFS file to an MVS sequential data set or partitioned data set member. You can specify text or binary data, and select code page conversion.
OGETX	Copy one or many files from a directory to a partitioned data set, a PDS/E, or a sequential data set. You can specify text or binary data, select code page conversion, allow a copy from lowercase filenames, and delete one or all suffixes from the filenames when they become PDS member names.
OPUT	Copy an MVS sequential data set or partitioned data set member to an HFS file. You can specify text or binary data, and select code page conversion.
OPUTX	Copy one or many members from a partitioned data set, PDS/E, or a sequential data set to a directory. You can specify text or binary data, select code page conversion, specify a copy to lowercase filenames, and append a suffix to the member names when they become filenames.
OSTEPLIB	Build a list of files that are sanctioned as valid step libraries for programs that have the set-user-ID or set-group-ID bit set. To use this command, you must be a superuser.
UNMOUNT (or UMOUNT)	Remove a file system from the file hierarchy. To use this command, you must be a superuser.

For information about existing TSO/E commands that you may commonly use, see *OS/390 TSO/E Command Reference*.

To get online help for using the OpenMVS TSO/E commands, you can use either the TSO/E OHELP or HELP command. For more detailed information on OHELP, see “Using the OHELP Command” on page 89. See “Entering a TSO/E Command” for information on entering TSO/E commands in TSO/E, the shell, and ISPF.

Entering a TSO/E Command

How you can enter a TSO/E Command depends on whether you are using the OMVS terminal interface or the asynchronous terminal interface you get with **rlogin**, **telnet**, or the Communications Server.

OMVS Terminal Interface: You can enter a TSO/E command:

- At the TSO/E `READY` prompt.

- In the shell, using the **tso** shell command. For more information on this command, see *OS/390 UNIX System Services Command Reference*.
- In the shell, by typing a TSO/E command at the shell prompt and pressing the TSO function key to run it.
- On an ISPF panel.

CAUTION:
You need to be aware of two things about entering TSO/E commands in ISPF:

- **On most ISPF panels, you must type TSO before the name of the TSO/E command; for example,**

  ```
  TSO OHELP 3 fopen
  ```

  However, on the TSO Command Processor panel (ISPF option 6), you can just enter the name of the TSO/E command, unless the command exists in both ISPF and TSO (for example, HELP or PRINT).
- **On most ISPF panels, ISPF folds what you type to uppercase. ISPF folds lowercase or mixed-case filenames to uppercase, even if they are enclosed in single quotes. However, the TSO Command Processor panel (ISPF option 6) processes what you enter exactly as it is typed—mixed case, uppercase, or lowercase.**

Asynchronous Terminal Interface: You can enter TSO/E commands in the shell, using the **tso** shell command. For more information on this command, see *OS/390 UNIX System Services Command Reference*.

Using a Relative Pathname on TSO/E Commands

If you run a TSO/E command by using the OMVS TSO subcommand or function key or the **tso -o** command, the TSO/E command runs in your TSO/E address space. The working directory of your TSO/E address space is typically your home directory. Therefore, if you specify a relative pathname on a TSO/E command, the system searches for it in your home directory—even if you are working in a different directory.

If you run a TSO/E command by using the **tso -t** command, it runs in its own process. If you run the command using a relative pathname, the system searches for it in your working directory.

Using the ISPF Shell to Work with Directories and Files

If you are a user with an MVS background, you may prefer to use the ISPF shellpanel interface instead of shell commands or TSO/E commands to work with the file system. The ISPF shell also provides the administrator with a panel interface for setting up users for OS/390 UNIX access, for setting up the root file system, and for mounting and unmounting a file system. For more information about the ISPF shell, see “Using the ISPF Shell” on page 168.

Using the Network File System Feature

Using the Network File System feature, you can mount HFS files on an empty directory at your workstation.

To access the hierarchical file system, you first enter the **mvslogin** command, which gives you permission to use NFS.

Then you enter the **mount** command to make a connection between a mount point on your local file system and a directory or file in the hierarchical file system. After a directory is mounted, you can create, delete, read, or write to a file in or below that directory in the file hierarchy; generally, you can treat a file in or below that directory as a member of your own workstation file system.

- For text files, the Network File System feature handles conversion between the EBCDIC code page used in the OS/390 shell and the ASCII code page used at your workstation.
- RACF checks the authority of a workstation user to access HFS files at the host. This is based on the authority of the MVS user ID specified on the **mvslogin** command.

Locking

Locking is local to the system you are working on; it is coordinated with other local users. If remote users are accessing a file at the same time as local users, locking is not coordinated between local and remote users.

External Links

An external link is a type of symbolic link that you can use to associate an MVS data set or PDS member with an HFS pathname. The external link lets the NFS client user transparently access an MVS data set using a pathname. A program using the **exec()** family of functions or the BPX1EXC (exec), BPX1LOD (loadhfs), or BPX1SPN (spawn) callable services can also access an MVS data set using an external link.

The data set appears in a mounted HFS directory with HFS files. If you are working with both MVS data sets and HFS files at the workstation, with an external link you can have one directory for both the data sets and the files—for example, **/host**, instead of **/host/ds** for the data sets and **/host/hfs** for the files.

For information on how to create an external link when working at the host, see “Creating an External Link” on page 218.

Security for the File System

This book assumes that your enterprise is using the Resource Access Control Facility (RACF). You could use an equivalent security product.

Power Failures and the File System

Should there be a power failure, you might lose recent data that is still buffered, but the file system structures, directories, inodes and such, will not be damaged. A shadow writing technique is used to ensure structural changes are always committed atomically. The HFS does its own repair, as needed, on each mount of a file system. This is based on records it keeps of changes in progress.

There is no **fsck** commandand the HFS was designed so that it is not needed. The **fsck** utility generally ensures structural integrity, not data integrity.

Of course, there is always a possibility that user data, critical file system data, or the media can be damaged, so prudent backup procedures are always warranted.

Chapter 15. Working with Directories

This chapter covers the following topics:
- The working directory
- Displaying the name of your working directory
- Changing directories
- Creating a directory
- Removing a directory
- Listing directory contents
- Comparing directory contents
- Finding a directory or file

The Working Directory

The shell always identifies a particular directory within which you are assumed to be working. This directory is known as the *working directory* (also known as the *current working directory*). To work with a file within your working directory, you need specify only the filename with a command. If you want to work with a file in another directory, you can change your working directory, using the **cd** shell command and naming the new directory. (Instead of changing directories, you could use relative notation to access a file in a different directory; see “Using Notations for Relative Pathnames” on page 206 for more information.)

When you type the OMVS command and begin working in the shell environment, you are first placed in your *home directory* as your working directory.

Displaying the Name of Your Working Directory

To check on the name of the directory you are currently working in, just enter the **pwd** command (print working directory).

If Alice Smith is working in her home directory, for example, the system displays the name of her working directory in this form:

```
/u/smitha
```

/u/smitha is the *pathname* of her working directory.

If Alice Smith enters the command `cd projecta`, the **projecta** subdirectory of her home directory becomes her working directory. If she issues the **pwd** command, it displays:

```
/u/smitha/projecta
```

Note: A directory name can be specified in two ways, with or without a trailing slash; for example:
/u/smitha/projecta
/u/smitha/projecta/

In this book, a trailing slash is not used.

Changing Directories

Use the **cd** command to change from one working directory to another. If you have permission to access the directory, you can move to any directory in the file system by using **cd** and the pathname for the directory:

```
cd pathname
```

See Chapter 17 for more information on directory permissions.

When you want to go to your home directory, just enter the **cd** command with no arguments:

```
cd
```

To change to a directory other than your home directory, you must supply the pathname. For example, if Alice Smith is working in her home directory (**smitha**) and she wants to switch to her **projectb** directory, she types the relative pathname:

```
cd projectb
```

To check that she has changed directories, Alice types **pwd** and the system displays:

```
/u/smitha/projectb
```

Using Notations for Relative Pathnames

To change directories quickly or to work with a filename in another directory, use these relative pathname notations:

dot notation (**.** and **..**)
tilde notation (~)

Dot Notation

If you use the **ls -a** command to list the contents of a directory, you see that every directory contains the entries **.** (dot) and **..** (dot dot):

. (dot) This refers to the working directory.

.. (dot dot)
This refers to the parent directory of your working directory, immediately above your working directory in the file system structure.

If one of these is used as the first element in a relative pathname, it refers to your working directory. If **..** is used alone, it refers to the parent of your working directory.

For example, if you are working in **/bin/util/src**, you can go to **/bin/util** by entering:

```
cd ..
```

Tilde Notation

A ~ (tilde) can be used from the OS/390 shell in several forms:

Notation	Meaning

~	Your home directory (that is, the directory given by your **HOME** environment variable). The command:
	`cp ~/file1 file2`
	
	copies **file1** in your home directory into **file2** in your working directory. This works regardless of what your working directory is.
	`cp file1 ~/dir`
	
	copies **file1** from the working directory into **dir** in your home directory.

~+	The variable **$PWD** (which contains the name of your working directory).
~–	The variable **$OLDPWD** (which gives the name of the working directory you were in immediately before the last **cd** command).
~*login name*	That user's home directory. For example: `cat ~allane/.profile` displays the profile file of **allane**, from that user's home directory. This is useful if there are a group of you working on a project and you have read-write access to some of each other's files. **Note:** In the OS/390 shell, your *login name* is your TSO/E user ID.

Example

For example, suppose that your home directory is **/u/turbo** and you are working in **/u/turbo/prog/src** and you want to display the file **limits** in the directory **/u/turbo/appl/hdr**. You could refer to the file in several different ways:

```
cat ../../appl/hdr/limits
cat ~/appl/hdr/limits
cat /u/turbo/appl/hdr/limits
```

Creating a Directory

Using the Shell:

To create a new directory, enter:

`mkdir pathname`

For example, if Alice Smith is working in her home directory, **smitha**, and she wants to create a new directory, **projecta**, *under her working directory,* she would enter:

```
mkdir projecta
```

The default mode (read-write-execute permissions) for a directory created with **mkdir** is:

owner=rwx
group=rwx
other=rwx

For directories, execute permission means permission to search the directory. The octal representation of these permissions is 777 (7 for the owner permission bits, the group permission bits, and the other permission bits).

The new directory, **projecta**, is one level below her working directory. Figure 25 on page 208 shows this relationship. If you do not specify an absolute pathname for

the directory to be created, the shell creates the new directory as a subdirectory of whatever your working directory is at the time you enter the command.

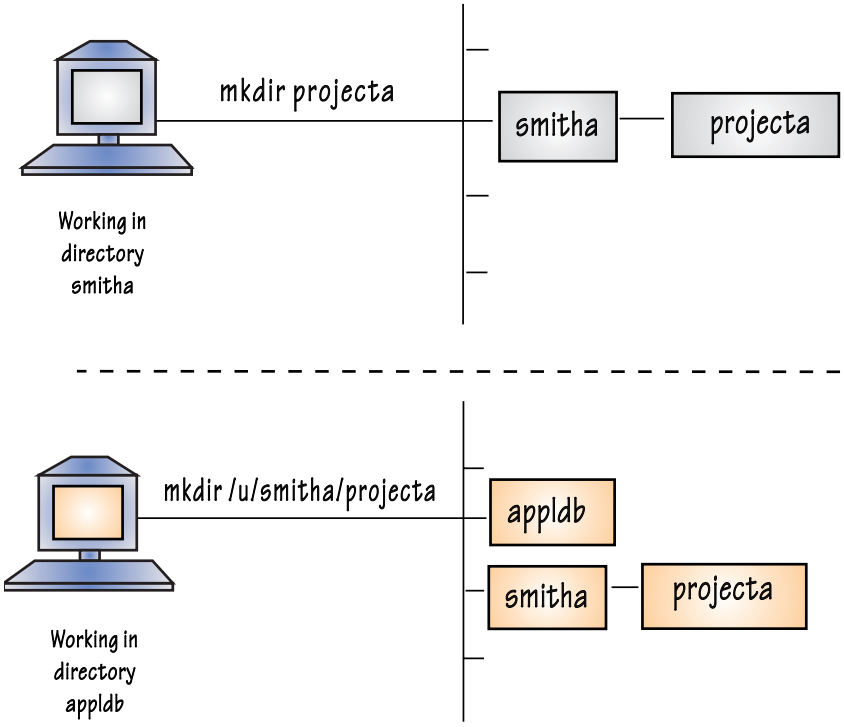


Figure 25. Creating a New Directory

If you want to create a new directory that is *not* under your working directory, specify an absolute pathname. Both directory names and filenames can be up to 255 characters long. You may want to adopt some naming convention that allows you to distinguish between directory names and filenames.

Your business may have adopted naming conventions for directories. For example, a typical convention is for each user to be assigned a directory based uniquely on the TSO/E user ID to make the name unique. Only that user would have write access to the directory. For information on how to change access permissions for a directory or file so that other users can read or write to it, see Chapter 17.

Using TSO/E:

To create a new directory, enter:

```
MKDIR 'directory_name' MODE(directory_permission_bits)
```

where directory_name specifies the pathname of the directory to be created; the pathname can be a full pathname or a relative pathname. Specify the name, which can be up to 1023 characters long, in single quotes. Specify MODE, the directory permission bits, in 3 octal characters; they can be separated by commas or blanks. The default mode (read-write-execute permission) is:

owner = rwx
group = r-x
other = r-x

The octal representation of these permissions is 755. (When MKDIR is used to create a directory, the default permission bits are different than when **mkdir** is used.) Here execute permission means permission to search the directory.

For example, to specify a directory with a full pathname and mode 700, enter:

```
MKDIR '/u/smitha/umods' MODE(7,0,0)
```

It is best to use a full pathname with the MKDIR command. When a relative pathname is specified, MKDIR defines the directory in the user's home directory, regardless of the working directory. If user Alice Smith is in her home directory **smitha** and wants to create a directory with a relative pathname and the default mode, then she can enter:

```
MKDIR 'umods'
```

The directory **umods** is one level below her home directory, **smitha**; its full pathname is **/u/smitha/umods**.

Removing a Directory

You can remove an empty directory (one with no files or subdirectories) from the file system with the **rmdir** command. The format of the command is:

```
rmdir directory
```

To remove your working directory, you must first move into another working directory.

To delete the files in a directory and the directory itself in one step, use the **rm** command with the **-r** option. The format of the command is:

```
rm -r file
```

where **file** is the name of the directory. Be careful! You may want to use the **-i** option so that you will be prompted to confirm the deletions:

```
rm -ri file
```

Listing Directory Contents

The **ls** command lists the contents of a directory. To see the contents of your working directory, enter:

```
ls
```

To list the contents of a different directory, add the relative or absolute name of the directory you want to look at, as in:

```
ls dira/dirb
ls abc/def/ghi
```

ls displays directory contents in alphabetic order. Typical **ls** output looks like:

```
bin         csrb.cpy    fifotest    makefl         temp.t
cc          etc         helplist    phones.com     totals
```

ls does not normally distinguish between directories, regular files, and special files. If you want a list of directory contents that does distinguish between file types, use the **-F** option. Entering:

```
ls –F
```

gives you output in the form:

```
bin/         csrb.cpy     fifotest|    makefl/         temp.t
cc/          etc/         helplist     phones.com*     totals/
```

The symbols following the filenames indicate the type of file:

/	Identifies a directory
*****	Identifies an executable file
|	Identifies a FIFO special file
@	Identifies a symbolic link
&;	Identifies an external link

If there is no character following the filename, the file is none of the above.

ls can list the contents of more than one directory at a time. For example:

```
ls dir1 dir2
```

lists the contents of the two given directories, one after the other. Try this command on a pair of directories to see what format **ls** uses.

The **ls** command with the **—E** option displays a character indicating whether or not the program is loaded from the shared library region. If the program is from the shared library region, an 'l' will appear as the fourth character in the second column. If the program is not from the shared library region, a '-' will appear.

Example

```
total 11
-rwxr-xr-x  -ps-    1 FRED  SYS1  101 Oct 02 16:30 james
-rwxrwxrwx  a-s-    1 FRED  SYS1  654 Oct 02 16:30 backup
-rwxr-xr-x  a---    1 FRED  SYS1   40 Oct 02 16:30 temp
-rwxr--r--  ap-l    1 FRED  SYS1  562 Oct 02 16:34 diag
-rwxr--r--  --sl    1 FRED  SYS1  106 Oct 02 16:53 bird
```

In the example above, the files *james*, *backup*, and *temp* are not loaded from the shared library region, but the files *diag* and *bird* are.

Comparing Directory Contents

You can use the command:

```
diff -r dir1 dir2
```

to check whole directories for change. With the **-r** option, **diff** compares the files in **dir1** with the files in **dir2** that have the same names.

This command can be useful if you have two directories that hold different versions of the same files and subdirectories.

You can use the **-r** option with other commands. For example:

```
cp -r dir1 dir2
```

copies all the files and subdirectories from **dir1** to **dir2**.

```
rm -r dir
```

removes all the files and subdirectories under **dir** and then removes **dir** itself.

Finding a Directory or File

The **find** command lists the names of all the files under a directory with a given characteristic or set of characteristics. The simplest version of the command is:

```
find dirname
```

which displays the names of all files under the given directory, including files in subdirectories under the directory.

```
find dirname –name pattern
```

displays the names of all files whose names have the form specified in *pattern*. For example,

```
find abc –name '*.lst'
```

lists the names of all files under the directory **abc** with the filename extension **.lst**. (The asterisk (*****) is a wildcard character that stands for any sequence of zero or more characters.) Using **find**, you can locate files quickly, even when you have a complicated file system structure, with many directories and subdirectories. See more information on the **find** command in *OS/390 UNIX System Services Command Reference*.

Chapter 16. Working with Files

This chapter covers the topics:
- Using an editor to create a file
- Naming files
- Deleting a file
- Deleting files over a certain age
- Identifying a file by its inode number
- Creating links
- Deleting links
- Renaming or moving a file or directory
- Comparing files
- Sorting file contents
- Counting lines, words, and bytes in a file
- Searching files by using pattern matching
- Browsing files
- Simultaneous access to a file
- Backing up and restoring files
- Listing process IDs of processes with open files

Using an Editor to Create a File

When logged into the shell, you have a choice of editors to use to create and change files, depending on which terminal interface you are using, OMVS or the asynchronous terminal interface. For details about the editors, see Chapter 18.

If you are using NFS from your workstation, you can directly edit HFS files with your editor of choice.

When you create directories and files, you can control access to them. Whenever you want, you can change the *access permissions* that are set when you first create a directory or file. See Chapter 17 for more information on access permissions.

Naming Files

A filename can be up to 255 characters long. To be portable, the filename should use only the characters in the POSIX portable filename character set:
- Uppercase or lowercase A to Z
- Numbers 0 to 9
- Period (.)
- Underscore (_)
- Hyphen (-)

Do not include any nulls or slash characters in a filename.

Doublebyte characters are not supported in a filename and are treated as singlebyte data. *Using doublebyte characters in a filename may cause problems.* For instance, if you use a doublebyte character in which one of the bytes is a . (dot) or / (slash), the file system treats this as a special delimiter in the pathname.

The shells are case-sensitive, and distinguish characters as either uppercase or lowercase. Therefore, **FILE1** is not the same as **file1**.

A filename can include a suffix, or *extension,* that indicates its file type. An extension consists of a period (`.`) and several characters. For example, files that are C code could have the extension **.c**, as in the filename **dbmod3.c**. Having groups of files with identical suffixes makes it easier to run commands against many files at once.

Processing in Uppercase and Lowercase

Case-sensitive processing means that an environment distinguishes and handles characters as either uppercase or lowercase. Therefore, **FILE1** is not the same file as **file1**. The availability of case-sensitive processing depends on the environment:

Shell Case-sensitive. In the file system, you can use mixed-case pathnames.

ISPF To issue a TSO/E command with an HFS pathname and get case-sensitive processing of the pathname, enter the command on a command line that supports mixed-case processing, for example the "Command Processor" panel (usually ISPF option 6). Some ISPF option panels convert the command and filename to uppercase before they are processed.

The default ISPF edit profile usually folds to uppercase the data you enter in a file. To prevent this, type `caps off` on the command line before you begin working in the file. After you enter `caps off`, it remains in your profile.

If you are working on a file and realize that you have been typing in uppercase when you really wanted lowercase, you can change the contents of the file to all lowercase. Type this on the command line:

```
c all p'>' p'<'
```

TSO/E Case-sensitive. Follow the syntax rules of the command you are using. For instance, make sure to enclose a pathname in single quotes when using commands such as ALLOCATE, OPUT, and so on.

JCL Case-sensitive. You can specify HFS files in DD statements by giving the absolute pathname (no relative pathnames) and enclosing the names in single quotes. Be careful to keep JCL keywords such as DD, PATH, and so on, in uppercase.

Note: Traditional MVS utilities may define their own requirements for allowing mixed-case filenames to be specified as input (as compared with the rules for specifying mixed-case filenames on DD statements in JCL). For example, you need to use the binder's CASE=MIXED option if you want to bind a load module into the file system and give the load module a lowercase name.

Deleting a File

The command **rm** can delete, or "remove", several files at once. For example:

```
rm file1 file2 file3
```

removes all the specified files.

Suppose Alice Smith's directory **projectb** had several old meeting notices in it that she wanted to delete: **0607.mtg**, **0615.mtg**, **0623.mtg**, and **0628.mtg**. She could remove all four with just a single command:

```
rm 06*.mtg
```

Be careful when using the wildcard asterisk (*****) for removing files; you may want to use the **-i** option, which prompts you to verify the deletion.

For the tcsh shell, see“Displaying Deletion Verification” on page 65for more information on how to control the wildcard asterisk.

Deleting Files over a Certain Age

The **skulker** shell script provides a way for the user to delete files based on when the file was last accessed. This can be useful if temporary files created by utilities, or files that were intended to be temporary but are forgotten about, need to be removed.

The **skulker** script is an OS/390 shell script and can be easily modified to fit any particular system or user needs. The script is located in **/samples**, but the system administrator should have relocated it somewhere else. Users should check with their system administrator for the location of the script. The script should be copied into the user's home directory or subdirectory, where it can be modified by the user if different removal criteria are desired.

It is also possible to invoke the **skulker** script with the **cron** daemon so that it may be run on a regular basis.

The format for running the **skulker** script is as follows:

```
skulker [-irw] [-l logfile] directory days_old
```

directory specifies the directory from which to delete files.

days_old specifies the age of files you want to remove, based on when the file was last accessed.

The **-i** option displays the files to be deleted and then prompts the user to terminate the script or continue with the deletion.

The **-r** option recurses subdirectories, removing both files and subdirectories that are equal to or older than the specified number of days.

The **-w** option does not delete the files, but sends warnings to the owner of each file (via **mailx**) that the file is a candidate for deletion.

The **-l** option allows the user to specify a *logfile* for listing the files that were deleted (or, in the case of the **-w** option, warnings that were sent) and any errors that might have occurred.

For more information on the **skulker** script, see *OS/390 UNIX System Services Command Reference*.

Identifying a File by Its Inode Number

In addition to its filename, each file in a file system has a identification number called an *inode number* that is unique in its file system. The inode number refers to the physical file, the data stored in a particular location. A file also has a device number, and the combination of its inode number and device number are unique throughout all the file systems in the hierarchical file system.

A directory entry joins a filename with the inode number that represents the physical file.

To display the inode numbers of the files in your working directory, just enter:

```
ls -i
```

If Alice Smith issues that command for her **proja** directory, she sees the following display:

```
1077 inspproc   1077 isoproc    1492 kgnproc   1500 mcrproc
```

Because the files **inspproc** and **isoproc** are hard-linked, they have the same inode number.

Creating Links

A *link* is a new pathname, or directory entry, for an existing file. The new directory entry can be in the same directory that holds the file or in a different directory. You can access the file under the old pathname or the new one. After you have a link to a file, any changes you make to the file are evident when it is accessed under any other name.

You might want to create a link:

- If a file is moved and you want users to be able to access the file under the old name.
- As an alias: You can create a link with a short pathname for a file that has a long pathname.

You can use the **ln** command to create a hard link or a symbolic link. A file can have an unlimited number of links to it.

Creating a Hard Link

A *hard link* is a new name for an existing file. You cannot create a hard link to a directory, and you cannot create a hard link to a file on a different mounted file system.

All the hard link names for a file are equally important as its original name. They are all real names for the one original file. To create a hard link to a file, use this command format:

```
ln old new
```

Thus, *new* is the new pathname for the existing file *old*. In Figure 26 on page 217, **/u/benson/proja** is the new pathname for the existing file **/u/smitha/proja**.

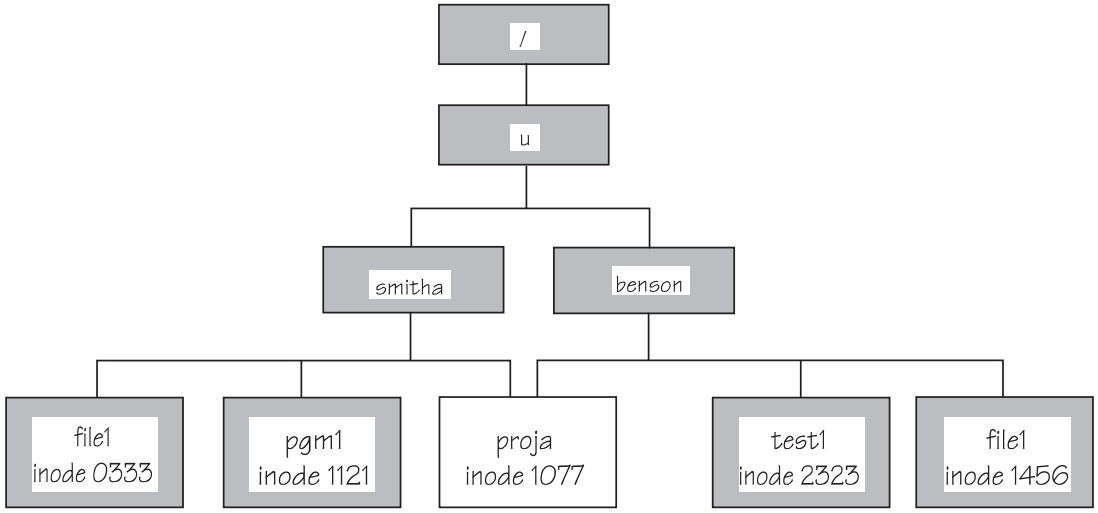


Figure 26. A Hard Link: A New Name for an Existing File. The hard link has an identical inode number.

When you create a hard link to a file, the new filename shares the inode number of the original physical file, as shown in Figure 26. Because an inode number represents a physical file in a specific file system, you cannot make hard links to other mounted file systems.

Creating a Symbolic Link

You can create a symbolic link to a file or a directory. Additionally, you can create a symbolic link across mounted file systems, which you cannot do with a hard link. A *symbolic link* is another file that contains the pathname for the original file—in essence, a reference to the file. A symbolic link can refer to a pathname for a file that does not exist.

To create a symbolic link to a file, use this command format:

```
ln -s old new
```

Thus *new* is the name of the new file containing the reference to the file named *old*. In Figure 27 on page 218, **/u/benson/proja** is the name of the new file that contains the reference to **/u/smitha/proja**.

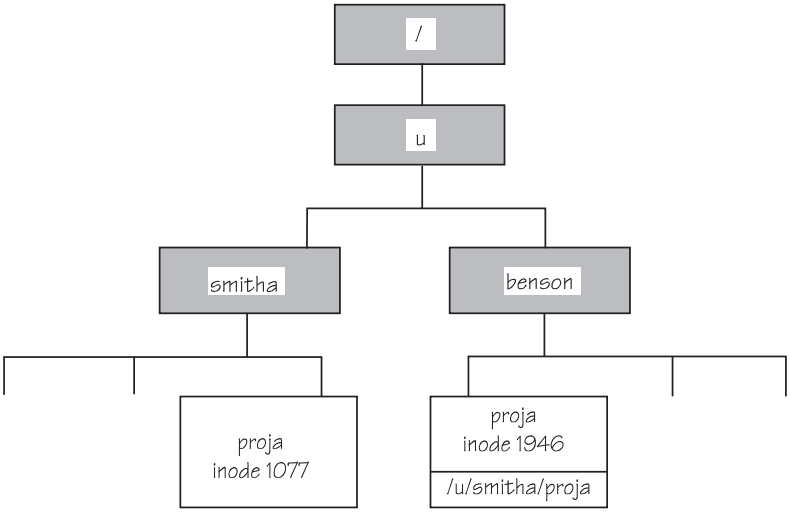


Figure 27. A Symbolic Link: A New File. A symbolic link has its own inode number.

When you create a symbolic link, you create a new physical file with its own inode number, as shown in Figure 27. Because a symbolic link refers to a file by its pathname rather than by its inode number, a symbolic link can refer to files in other mounted file systems.

To understand how a symbolic link that is a component of a pathname is handled during pathname resolution, see “Resolving a Symbolic Link in a Pathname” on page 199.

Creating an External Link

An *external link* is special type of symbolic link, a file that contains the name of an object outside of the hierarchical file system. Using an external link, you associate that object with a pathname. For example, **setlocale()** searches for locale object files in the HFS, but if you want to keep your locale object files in a partitioned data set, you can create an external link in the HFS that points to the PDS. This will improve performance by shortening the search that **setlocale()** makes.

A file can be an external link to a sequential data set, a PDS, or a PDS member. When a file is an external link to an MVS data set, an NFS client user can use the pathname to access the data set. You must be using the Network File System feature to use the pathname to edit, browse, or display the attributes of the data set that is the target of an external link. Working in a shell, you can create (**ln**) an external link, display information (**ls**) about the link (not the target of the link), or delete (**rm**) the link.

These services support external links:

- NFS client: You can create external links as files within the HFS, and then access these files as an NFS client user to access the MVS datasets that they point to.
- A program using the **exec()** family of functions, the BPX1EXC (exec) callable service, the BPX1LOD (loadhfs) callable service, or the BPX1SPN (spawn) callable service can access an MVS data set using an external link. This capability includes external link programs that are invoked as commands in the shell.
- Dynamic link libraries: The external link name used on a DLL load is a member name. For example, you would code a link as:

```
ln -e IMWYWWS /usr/lpp/internet/bin/wwwss.so
```

where IMWYWWS is the member name linked to the file **wwwss.so**.

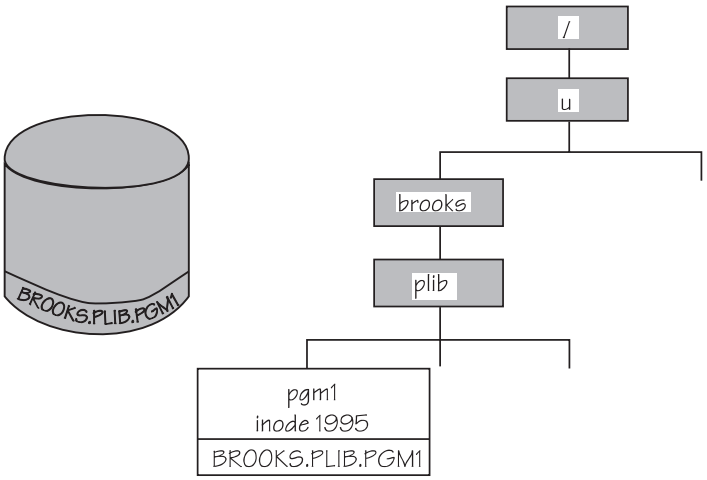


Figure 28. An External Link: A New File. An external link has an inode number. The MVS data set does not.

To create an external link to a data set, use this command format:

```
ln -e old new
```

In Figure 28, **/u/brooks/plib/pgm1** is the name of the new file that contains the reference to the partitioned data set BROOKS.PLIB.PGM1.

Limitations of an External Link: OS/390 UNIX C programs running cannot **fopen()** or **fread()** an external link. For more information, look under the **ln** command in *OS/390 UNIX System Services Command Reference*.

Deleting Links

To delete a file that has hard links, you must enter **rm** against all the link names, including the original file name. If you try to delete a file that is hard-linked, its contents do not disappear until you remove every link to it.

To delete a file that is a symbolic link, you enter **rm** against the symbolic link name. This removes the link, not the file it refers to. When you delete a file that is symbolically linked, any remaining symbolic links refer to a file that no longer exists. If you know the names of the symbolic link files, you may want to delete them.

To delete a file that is an external link, run **rm** against the external link name. If you delete a data set that is externally linked, the remaining external link refers to a data set that no longer exists.

Renaming or Moving a File or Directory

You can use the **mv** command to *move* or rename files. For example:

```
mv file1 file2
```

moves the contents of **file1** to **file2** and deletes **file1**. This is similar to:

```
cp file1 file2
rm file1
```

except that, when the files are in the same mountable file system, **mv** renames the file rather than copying it. **file1** and **file2** do *not* have to be in the same directory.

The **mv** command can move several files from one place to another.

For example:

```
mv file1 file2 file3 directoryb
```

moves all three files to **directoryb**.

Using the **-R** or **-r** option, you can move a directory and all its contents (files, subdirectories, and files in subdirectories) into another directory. For example:

```
mv -R directorya directoryb
```

Comparing Files

Consider the following situation: A warehouse has an *active file* that keeps track of current inventory. As goods are brought in, appropriate records are added to the file. As orders are shipped out, the records are deleted. At the end of the day, the warehouse makes a copy of the active file to keep as a permanent journal.

It would be useful for such a business to be able to compare one day's journal to another day's to see what has changed. This can be done with the **diff** command:

```
diff oldfile newfile
```

compares the two files. The output of **diff** shows lines that are in one file but not in the other. The lines in *oldfile* but not in *newfile* are displayed with a < in front of them. Lines in *newfile* but not in *oldfile* are displayed with > in front.

For example, say you have a file **wmnhist.text** with one line in it:

```
Susan B. Anthony awoke one morning
```

Then you created a copy of the file with the command:

```
cp wmnhist.txt newhist.txt
```

You use an editor—either the ISPF editor or the **ed** text editor—to change the first line in **newhist.text** to:

```
Sojourner Truth awoke one morning
```

You save the file. Now you enter the command:

```
diff wmnhist.txt newhist.txt
```

diff displays:

```
1c1
< Susan B. Anthony awoke one morning
---
> Sojourner Truth awoke one morning
```

The `1c1` at the beginning of the **diff** output indicates that line `1` in the old file has `c`hanged (`c`) when compared with line `1` in the new file. **diff** shows what must be changed in the first file to make it look like the second file. Remember this sequence when you look at the output of **diff**. Here the first file, **wmnhist.txt**, contained the line `Susan B. Anthony awoke one morning` where the second file, **newhist.txt**, has `Sojourner Truth awoke one morning`.

New lines are indicated with an `a` (add lines), and lines that should be deleted are indicated with a `d` (delete). See *OS/390 UNIX System Services Command Reference* for more details.

diff helps you determine what has changed in the time that elapsed between saving the two files. The same sort of operation is useful in many record-keeping situations, any time you have two different versions of the same file and you want to check the differences.

Sorting File Contents

When you create a file of records, you usually do not type the information in any particular order. However, you may want to keep lists in some useful order after you have entered the information. To sort the records in a file, use the **sort** command. **sort** assumes two things:

- Your file contains one record per line. To put it another way, there is a single <newline> character between a record and the next record.
- The fields in a record are separated by recognizable characters. In the sample file **comics.lst** in **/samples** (shown in Figure 29 on page 222), we use colons.

```
Detective Comics:572:Mar:1987:$1.75
Demon:2:Feb:1987:$1.00
Ex-Mutants:1:Sep:1986:$2.60
Justice League of America:259:Feb:1987:$1.00
Boris the Bear:1:Sep:1986:$1.50
Flaming Carrot:14:Oct:1986:$2.75
Demon:4:Apr:1987:$1.00
The Question:1:Jan:1987:$2.10
Elektra:7:Feb:1987:$2.00
Howard the Duck:29:Jan:1979:$0.35
Wonder Woman:3:Apr:1987:$1.00
Justice League of America:261:Apr:1987:$1.00
Secret Origins:10:Jan:1987:$1.75
The Question:2:Mar:1987:$2.10
Justice League of America:258:Jan:1987:$1.00
Batman:566:Sep:1986:$1.00
Legends:3:Jan:1987:$1.00
Daredevil:234:Sep:1986:$0.95
Legends:5:Mar:1987:$1.00
Daredevil:237:Dec:1986:$0.95
Star Trek:29:Aug:1986:$0.95
Green Lantern Corps:203:Aug:1986:$0.95
The Shadow:3:Jul:1986:$2.10
Green Lantern Corps:204:Sep:1986:$1.00
Son of Ambush Bug:3:Sep:1986:$1.00
New Teen Titans:26:Dec:1986:$2.10
Legends:1:Nov:1986:$1.00
Detective Comics:568:Nov:1986:$1.00
Boris the Bear:3:Dec:1986:$2.30
Cerebus:89:Aug:1986:$2.00
Legends:4:Feb:1987:$1.00
Swamp Thing:57:Feb:1987:$1.00
Wonder Woman:1:Feb:1987:$1.00
Flaming Carrot:13:Jul:1986:$2.00
Ex-Mutants:2:Oct:1986:$2.60
Ex-Mutants:3:Dec:1986:$2.75
Flaming Carrot:12:May:1986:$2.00
Midnite Skulker:2:Aug:1986:$2.50
Strikeforce Morituri:2:Jan:1987:$0.95
Strikeforce Morituri:1:Dec:1986:$0.95
Demon:3:Mar:1987:$1.00
Watchmen:5:Jan:1987:$2.10
Watchmen:6:Feb:1987:$2.10
Watchmen:7:Mar:1987:$2.10
Watchmen:8:Apr:1987:$2.10
Watchmen:4:Dec:1986:$2.10
Watchmen:3:Nov:1986:$2.10
Watchmen:1:Sep:1986:$2.10
Watchmen:2:Oct:1986:$2.10
Moonshadow:2:May:1985:$1.75
Moonshadow:3:Jul:1985:$1.75
Border Worlds:1:Jul:1986:$2.80
Daredevil:239:Feb:1987:$0.95
Dark Knight:4:Oct:1986:$4.50
Firestorm:55:Jan:1987:$1.00
Dark Knight:1:Jul:1986:$4.50
Superman:2:Feb:1987:$1.00
Legends:2:Dec:1986:$1.00
Cerebus:87:Jun:1986:$2.00
Swamp Thing:54:Nov:1986:$1.00
Son of Ambush Bug:6:Dec:1986:$1.00
Bozz Chronicles:2:Feb:1986:$1.75
Bozz Chronicles:3:May:1986:$1.75
```

Figure 29. A Sample File: comics.lst

To sort a file such as our comic book file, enter:

```
sort /samples/comics.lst
```

This command sorts the list and displays it. To save the sorted list in a file, enter:

```
sort /samples/comics.lst >filename
```

where *filename* is the name of the file where you want to store the sorted list. For example:

```
sort /samples/comics.lst >sorted.lst
```

sorts the file and stores the result in **sorted.lst** without changing the input file.

When you use `>filename` to redirect sorted output into a file, you may want to make the output filename different from the (unsorted) input filename. If you want to overwrite a file with its sorted contents, see the description of the **–o** flag in **sort** in *OS/390 UNIX System Services Command Reference*.

Using Sorting Keys — an Example

By default, **sort** sorts according to all the information in the record, in the order given in the record. Since the name of the comic book is the first thing on the line, the output is sorted according to comic book name. But suppose that you want to sort according to some different piece of information. For example, suppose you want to sort by date of publication. You can do this by specifying sorting keys.

A *sorting key* tells **sort** to look at specific fields in a record, instead of looking at each record as a whole. A sorting key also tells what kind of information is stored in a particular field (for example, an ordinary word, a number, or a month) and how that information should be sorted (in ascending or descending order).

A sorting key can refer to one or more fields. Fields are specified by number. The first field in a record is field number 1, the field after the first separator character is field number 2, and so on. In the comic book list, the month is field number 3, and the year is field number 4.

A single **sort** command can have several sorting keys. The most important sorting key is given first; less important sorting keys follow. Let us look at an example that sorts by year and then by month within a year. Therefore, the first sorting key refers to the year field, and the second to the month field. To specify a sorting key, use the **-k** option. This option has the following format:

```
-k start_field[.char1] [opts] [,end_field[.char2] [opts]]
```

where *start_field*, *end_field*, *char1*, and *char2* are all integers.

- *start_field* indicates which field in the input record contains the start of the sorting key.
- *char1* indicates which character in that field is the first character of the key. Omitting *char1* means the key begins with the first character of the starting field.

In our example, the first sorting key (referring to the year) has a *start_field* value of `4` (since the year is field 4). We do not need to specify *char1*, since we want to start the key with the first character of the year field.

The options, *opts*, are specified with letters; they identify the type of data in the specified field and tell how to sort it. Some of the possible options and their meanings are:

d Indicates that the field contains uppercase, lowercase, or mixed-case letters, letters and digits, or digits. **sort** sorts the field in *dictionary* order, ignoring all other characters.

M Indicates that the field contains the name of a month. **sort** looks only at the first three characters of the name, so `January`, `JAN`, and `jan` are all equal.

n Indicates that the field contains an integer (positive or negative).

Putting an `r` after any of these letters tells **sort** to sort in reverse order (from highest to lowest rather than lowest to highest). For example, `Mr` means to sort in the order `December`, `November`, `October`, and so on.

In our example the sorting key based on the year uses `n`. Thus, the sorting key for the year field (4) in the file **comics.lst** is:

```
-k 4n
```

The second sorting key in the example refers to the month field (3). This key has the form:

```
-k 3M
```

A **sort** command that uses sorting keys needs to know which character separates the record fields. You can specify this with the option **–t** followed by the separator character. The example uses **–t:**. Therefore, the full **sort** command is:

```
sort -t: -k4n -k3M  comics.lst >sorted.lst
```

The file to be sorted comes after the various options. This is the order that you must use. The redirection construct can come anywhere on the line, but is usually put at the end.

Counting Lines, Words, and Bytes in a File

The **wc** command tells you how big a text document is.

```
wc file file ...
```

tells you the number of lines, words, and bytes in each file.

If you want to find out how many files are in a directory, enter:

```
ls | wc
```

This pipes the output of **ls** through **wc**. Because **ls** prints one name per line when its output is being piped or redirected, the number of lines is the number of files and directories under your working directory.

Searching Files by Using Pattern Matching

One of the most common record-keeping operations is obtaining a sublist of a list. For example, you might want to list all the *Watchmen* comics that appear in the main comics list. The command to do this is **grep**.

The simplest form of the **grep** command is:

```
grep word file
```

where *word* is a particular sequence of characters that you want to find, and *file* is your list of records. **grep** lists every line in the file that contains the given word. For example:

```
grep Watchmen comics.lst
```

lists every line in **comics.lst** that contains the word `Watchmen`. As another example:

```
grep 1986 comics.lst
```

lists every line in **comics.lst** that contains the sequence of characters `1986`. Presumably, this lists all the comics that were published in 1986.

```
grep Jul:1986 comics.lst
```

lists all the comics published in July 1986.

If the string of characters you want to search for contains a blank, put single quotes (apostrophes) around the string; for example:

```
grep 'Dark Knight' comics.lst
```

You can save a sublist created by **grep** in a file using redirection:

```
grep Elektra comics.lst >el.lst
```

Patterns

The examples of **grep**, so far, have displayed the records in a file that contain the desired string anywhere in the line. If you want to be more specific—say to find records that *begin* with a certain string of characters (instead of having that string anywhere in the line)—use **grep** with *patterns* instead of strings.

To understand patterns, it helps to think about the special wildcard characters discussed in “Using a Wildcard Character to Specify Filenames” on page 80. Remember that you can use patterns in commands; for example:

```
rm *.txt
```

removes all files in the working directory that have the **.txt** extension. Instead of specifying a single filename, this example uses the special character * to represent any filename of the appropriate form.

In the same way, a **grep** pattern uses special characters so that one pattern can represent many different strings.

Note: The special characters for **grep** patterns are not the same as the characters used on command lines, and the mechanisms involved are also different: however, patterns and *wildcard characters* are conceptually similar.

Special characters used in a pattern are called *pattern characters,* or *metacharacters*. Some pattern characters are:

^ (caret)
: Stands for the beginning of a line. For example, **^abc** is a pattern that represents *abc* at the beginning of a line.

$ (dollar sign)
: Stands for the end of a line. For example, **xyz$** is a pattern that represents *xyz* at the end of a line.

. (dot or period)
: Stands for any (single) character. For example, **a.c** is a pattern that represents *a,* followed by any character, followed by *c*.

*** (asterisk)**
: Indicates zero or more repetitions of part of a pattern. For example, **.***

indicates zero or more repetitions of **.** (period). Since the **.** stands for any character, **.*** stands for any number of characters. For example, **^a.*z$** is a pattern that represents *a* at the beginning of a line, *z* at the end, and any number of characters in between.

A typical **grep** command has the form:

```
grep 'pattern' file
```

This displays all the records in the file that match the given pattern. For example:

```
grep '^Superman' comics.lst
```

displays all the records that begin with the word *Superman*.

```
grep '00$' comics.lst
```

displays all the records that end in *00*.

If you want to use the *literal* meaning of a pattern character instead of its special meaning, put a backslash (\) in front of the character. For example:

```
grep '\$1\.00$' comics.lst
```

finds all the lines that end in *$1.00*. Without a backslash in front of the **$** and **.** (period), they would have their special pattern meanings.

Regular Expression

More complex patterns than the ones discussed here are accepted. The formal name for a pattern is a *regular expression*. For further information, see the appendix on regular expressions in *OS/390 UNIX System Services Command Reference*.

Browsing Files

When you display, or "browse," a file, you cannot make any changes to the file while you are viewing it. You can browse an HFS file using ISPF or using shell commands. With shell commands, you have the choice of browsing the file in an unformatted or formatted display.

Browsing Files without Formatting

Using the Shell:

The OS/390 shell has a quick way to find out what is in a given file: the **head** command and the **tail** command.

head *filename*
: Displays the first 10 lines of the given file or files.

tail *filename*
: Displays the last 10 lines of the given file or files.

Suppose you have a file that contains records sorted according to date. **tail** tells you the date of the last records in the file, giving you an idea of how current the file's contents are. In a sorted comic book list, for example, **tail** could show the most recent comics that had been recorded in the file.

To display the contents of an entire file, you can use any of these commands: **cat**, **pg**, **more**, or **obrowse**.

Using ISPF:

1. To use ISPF to browse an HFS file, you can do either of the following:
   - Enter the TSO/E OBROWSE command, which accesses ISPF, in either of two formats:
     – OBROWSE with no filename. This displays the Browse - Entry panel (see Figure 30).
     – OBROWSE followed by the pathname for the file. This command displays the file, which you can begin browsing.

       If you are working in the shell and you enter OBROWSE with a relative filename, OBROWSE searches for the file in your home directory. This is because in this situation the relative pathname is relative to the current working directory of the TSO session, usually your home directory.
   - Select an option for browsing HFS files on the ISPF menu, if such an option is available. This displays the Browse Entry panel; see Figure 30.
2. Complete the Browse Entry panel, if it is displayed.
   These are the fields on the panel:

```
-------------------- BROWSE - ENTRY PANEL ---------------------------
Command ===>

Directory      ===> .


Filename       ===>

If data is to be browsed as fixed length records:
 Record length ===>         Leave blank to browse delimited files
```

Figure 30. ISPF Browse Entry Panel for an HFS File

Directory
Type the directory name.

Filename
Type the filename.

Record length
To specify fixed-length browse, type the record length: The range is 10 to 32 760 characters.

To specify variable-length records, which are delimited by a <newline> character, leave this field blank.

3. After the file is displayed, you can use function keys to scroll forward and backward in the file.

For complete information about browsing, see *ISPF/PDF Guide and Reference*.

Browsing Files with Formatting

Using the Shell:

Formatting is controlling the appearance of the file contents when you browse or print them. You can use the **pr** command to browse (or “print to standard output”) a formatted file:

`pr` *`file`*

You can specify more than one filename, each separated from the other by a space.

If you do not specify any options, **pr** formats the file into single-column, 66-line pages, each with a 5-line header. The first 2 lines are blank. On the 3rd line appear the file's pathname, the date of its last modification, and the current page number. The next 2 lines are blank, and the text of the file begins on the 6th line. At the end of each page, there are 5 blank lines. There are numerous options for the **pr** command; for example, you can specify the page number where the display is to begin, specify output in columns, or change the width of the displayed page.

Simultaneous Access to a File

It is possible that two or more utilities or programs could be accessing the same file at the same time, making changes. For example, two people using **ed** could edit the same file at the same time. When a file has been accessed by more than one user simultaneously, the last changes saved overwrite any previous changes.

In a program, you can use byte-range locking to avoid this problem. For more information about byte-range locking in a program, see *OS/390 C/C++ Programming Guide*.

For information on locking between local users and remote users using the Network File System feature, see "Locking" on page 204.

Backing Up and Restoring Files: The Options

There are several options for backing up and restoring files:

- Data Facility System-Managed Storage Hierarchical Storage Manager (DFSMShsm) provides automatic backup facilities for HFS data sets. The system programmer uses DFSMShsm facilities to back up mountable file systems by backing up the HFS data sets that contain them on a regular basis; the data sets can be restored when necessary. DFSMShsm is also used for migrating (archiving) and restoring unmounted file systems.
- ADSTAR Distributed Storage Manager (ADSM) provides a backup function for OS/390 UNIX clients. There are two types of backup: *incremental*, in which all new or changed files are backed up; and *selective*, in which the user backs up specific files.

  Backup can be performed automatically or when the user requests it. The user can initiate a specific type of backup or start the scheduler, which will run whatever action the ADSM administrator has scheduled for the user's machine.

  Information about using the OS/390 UNIX client is documented in:
  - *ADSM Using the UNIX Backup-Archive Clients*, SH26-4052.
  - *ADSM Installing the Clients*, SH26-4049.
- From the shells, you can manually back up data by using the TSO/E OGET command to copy files into an MVS sequential data set, partitioned data set, or partitioned data set extended (PDSE) that you know is backed up. To simplify archiving multiple files, the **pax** or **tar** utilities can be used to consolidate individual component files into a single archive file that can then be copied to an MVS data set. With Release 8 or later, **pax** and **tar** can write the archive directly to an MVS data set eliminating the need to manually copy the archive with OGET. For more information on using **pax** or **tar** and OGET to backup and restore file from the shell, see "Backing Up and Restoring Files from the Shell" on page 229.

The **cron** utility can be used to automatically start running **pax** or **tar** commands at a specified time.

After the files are in an MVS data set, you can load it to a tape. Conversely, you can load files from a tape into an MVS data set and then copy them into the file system. For more information, refer to “Transporting an Archive File on Tape or Diskette” on page 299.

Backing Up and Restoring Files from the Shell

This section describes how to use the **pax** or **tar** utilities to back up and restore files. The purpose of either utility is to store the data and attributes of one or more *component files* into a single file referred to as the *archive file*. **pax** is considered the standard utility for managing archive files, replacing **tar**, and so **pax** is used as the default utility in the examples that follow. However, **tar** is still widely used, and in the OS/390 environment provides practically equivalent function. So, the corresponding **tar** commands are also shown.

Both **pax** and **tar** support multiple archive formats and options that allow a greater or lessor degree of file characteristics to be preserved. The USTAR format allows the most information to be saved so it is used as the default format in the examples that follow. For more information about the USTAR and other archive formats, refer to *OS/390 UNIX System Services Command Reference*. Because both **pax** and **tar** can read and write archives in USTAR format, either utility can be used to restore an archive created by the other. The significant difference between the two utilities is that only **pax** can perform code page conversion on files during creation of, or extraction from, an archive. Users of **tar** can use the **iconv** utility to perform the same conversion on files as a separate step.

Both **pax** and **tar** support inline compression and decompression of files. Because compressed archives occupy an average of 50-60% percent of the uncompressed archive, many of the examples shown here use compression. Note that compressed archives are not guaranteed to be portable to other UNIX systems.

Archives can be copied to an MVS data set using the TSO/E OGET command and later copied back to the file system using the TSO/E OPUT command. For Release 8 and later, **pax** and **tar** can read and write archives that reside in an MVS data set making it unnecessary to first manually move files between the file system and MVS using OGET or OPUT.

For Release 9 and later, **pax** and **tar** will support file names and link names that exceed 100 characters in length. The utilities will remain compatible with other UNIX systems and with previous versions of OS/390 UNIX.

The remainder of this section describes the following specific steps for backing up and restoring files to and from an MVS data set and performing other related archive management tasks.

- Backing up a complete directory into an MVS data set
- Restoring a complete directory into an MVS data set
- Viewing the contents of an archive
- Converting between code pages
- Appending to an existing archive
- Storing selected files into an archive
- Restoring selected files from an archive

These examples demonstrate the most common tasks related to backing up and restoring files and do not attempt to describe all of the options of the **pax** and **tar** utilities. Refer to *OS/390 UNIX System Services Command Reference* for a complete description of **pax** and **tar**.

Backing Up a Complete Directory into an MVS Data Set

To back up the complete directory **/u/project**, including the subdirectories and their contents, into a compressed archive stored in the MVS data set `'PROJECT.ARCHIVE'`, enter the following commands:

```
cd /u/project
pax -wzvf /tmp/project.pax.Z  ./
tso "oget '/tmp/project.pax.Z'  'PROJECT.ARCHIVE' binary"
```

Notes:

1. For Release 8 and later, **pax** can write directly to the MVS data set and so the OGET command can be skipped by specifying the MVS data set on the **pax** command:

   ```
   pax -wzvf "//'PROJECT.ARCHIVE'" ./
   ```

2. The equivalent **tar** commands are:

   ```
   tar -cUzvf /tmp/project.pax.Z  ./
   ```

   To write directly to MVS (Release 8 or later):

   ```
   tar -cUzvf "//'PROJECT.ARCHIVE'" ./
   ```

3. Changing to the current directory first is done to simplify the **pax/tar** command and so that the files are stored in the archive using a pathname relative to the current directory. This simplifies restoring the archive later to a different directory. The `"./"` is used rather than an asterisk to collect any component files that begin with `"."` in the current directory.
4. The archive is written to a directory that is not in the source path being archived in order to prevent **pax/tar** from trying to store the archive within itself. Doing so can cause **pax/tar** to loop infinitely during creation or can result in corrupted files during restore.
5. Naming archives with a suffix of `"pax.Z"` (or `"tar.Z"`) is not required by **pax/tar**, but is done as a convention to identify them as **pax** or **tar** archive files. The `".Z"` is used to identify a compressed file.
6. The **—z** option is used to turn on compression and is not required.
7. The **—v** option is used to display the names of files as they are being stored and is not required.

Restoring a Complete Directory from an MVS Data Set

To restore the directory backed up in the previous example to **/u/project_old** enter the following commands:

```
tso "oput 'PROJECT.ARCHIVE' '/tmp/project.pax.Z  binary"
cd /u/project_old
pax -pe -rvf /tmp/project.pax.Z
```

Notes:

1. For Release 8 and later, **pax** can read an archive directly from an MVS data set and so the OPUT command can be skipped by specifying the MVS data set on the **pax** command:

   ```
   pax -pe -rvf  "//'PROJECT.ARCHIVE'"
   ```

2. The equivalent **tar** command is:

   ```
   tar -p -xvf /tmp/project.pax.Z
   ```

To read directly from MVS (Release 8 or later):

```
tar -p -xvf "//'PROJECT.ARCHIVE'"
```

3. The **—pe** option for **pax** and the **—p** option for **tar** are used to restore the original owner, group, modes, and extended attributes. If you do not have the appropriate privileges to restore these, warning messages are generated. These options are not required to restore the component files and can be omitted. For **tar**, the **—o** option is also used to disable restoring the owner and group.
4. **pax** and **tar** automatically detect the archive format and whether the archive is compressed, so the **—z** option for **pax** and, for **tar**, the **—U** option is not required. If these options are used, **pax/tar** fails if the archive is not compressed or not in USTAR format.
5. The **—v** (verbose) option is used to display the names of files as they are being restored and is not required.
6. Component files can be renamed during extraction by **pax** using the **—i** or **—s** option.

Viewing the Contents of an Archive

To view the contents of the **/tmp/project.pax.Z** archive created in the previous step, enter one of the following commands:

To list only the names of component files:

```
pax -f /tmp/project.pax.Z
```

To list the contents in a verbose format similar to "`ls -l`":

```
 pax -vf /tmp/project.pax.Z
```

For Release 7 and later, to list the extended attributes in a verbose format similar to "`ls -E`":

```
pax -Ef /tmp/project.pax
```

Note: The equivalent **tar** commands are:

- To list only component files: `tar -tf /tmp/project.pax.Z`
- For a verbose list: `tar -tvf /tmp/project.pax.Z`
- For extended attributes (Release 7 or later):

  ```
  tar -tEf /tmp/project.pax.Z
  ```

Converting Between Code Pages

Archives are often used to move files between UNIX systems. When an archive contains text files, it is frequently the case that the file must be converted from the source systems default code page to the target systems code page. This can be done using the **iconv** utility on each file before storing in an archive or after restoring from an archive. The **pax** utility, however, provides an inline code page translation option, **—o** that can simplify this task. For example:

- To convert component files from EBCDIC (IBM-1047) to ASCII (ISO8859-1) when storing them in an archive:

  ```
  pax -o to=iso8859-1 -wzvf /tmp/project.pax.Z  ./
  ```

- To convert component files from ASCII (ISO8859-1) to EBCDIC (IBM-1047) when extracting them from an archive:

  ```
  pax -o from=iso8859-1 -pe -rzvf /tmp/project.pax.Z
  ```

Notes:

1. The **—o** option allows both a ″from″ and ″to″ code page to be specified on the same command. If a ″from″ or ″to″ codepage is not specified, **pax** assumes it to be EBCDIC (IBM-1047).
2. For more information about the code sets supported for this command, see the Coded Character Set Conversion Table in *OS/390 C/C++ Programming Guide*.

Converting archives that contain text and non-text component files. Often, archives contain both text and non-text files. Examples of non-text files are image files such as JPGs and GIFs, and other **pax/tar** archives. When the **—o** option is specified, **pax** converts all files regardless of type. This corrupts non-text files. The general approach for overcoming this limitation is to run **pax** two or more times against the same archive, extracting components files in groups of text and non-text types. Whether it is easier to identify (by filename) text files or non-text files will determine how you approach this.

For example, suppose you wish to restore the archive ″mywebsite.pax″ that consists of HTML files (text files) and JPG files (JPEGS, non-text image files) and that was created on a system whose default code page is ASCII (ISO8859-1) into the directory **/u/website**. Assume that the majority of the files are HTML files and that the archived files represent several levels of sub-directories.

First restore the entire archive using the **—o** option.

```
pax -rvf mywebsite.pax  -o to=IBM-1047
```

This extracts and converts all component files. The extracted non-text JPEG files would be corrupted because they were also converted. The next step would be to re-extract the JPG files without the **—o** option. The **pax** option allows you to specify a ″pattern″ used to extract only those files that match the pattern. However, because of the multiple subdirectories, there is no way to create a pattern that would match every JPG in each subdirectory. Instead, a list of each file name to extract must first be created and then used as the pattern for the **pax** command to extract the files. Issuing the following command in the OS/390 shell would accomplish this:

```
pax -rvf mywebsite  $( pax -f mywebsite.pax | grep -i JPG$ )
```

The above command consists of two parts:

```
pax -rvf mywebsite  $(    )
```

and

```
pax -f mywebsite.pax | grep -i JPG$
```

The first part is simply the regular **pax** command for extracting files from an archive. The $( ) expression says to first run the command between the parenthesis and substitute the results in place. The second part is the command that generates a list of file names in the archive that end in ″JPG″ (or any mixed-case variation).

The above example is one approach. In general, for any archive, the breakdown of text to non-text files and the uniqueness of the names that identify each type dictate the manner and order in which the files are extracted. For example, we could have reversed the above process first extracting all files without using the **—o** option, and then re-extracting the HTML files on the second command using the **—o** option to convert the files.

Appending to an Existing Archive

Additional files and/or directories can be added to a previously created *uncompressed* archive using the **—a** (append) option. For example, to add the file ″oops.forgot″ to the existing archive ″allfiles.pax″:

```
pax -awvf allfiles.pax  oops.forgot
```

This adds ″oops.forgot″ to the end of the archive. If a file already exists in the archive with the same name, it will NOT be overwritten or replaced.

Notes:

1. Release 8 and above can append directly to archives in sequential MVS data sets only. **pax** and **tar** do not support appending to archives residing in partitioned MVS data sets.
2. The equivalent **tar** command is:

   ```
   tar -rvf allfiles.pax  oops.forgot
   ```

Backing Up Selected Files by Date

The following examples pertain to the OS/390 shell only, and demonstrate how to back up selected files that may have been modified within a specified number of days. The procedure for doing this is to create a ″find″ command that returns the list of files meeting the specified criteria and using the output from this command as the list of files input to **pax**.

The following example demonstrates how to back up all files in the directory **/u/source** that have been modified in the last week.

```
pax -wzvf backup.pax.Z    $( find /u/source -type f -mtime -8 )
```

The following example demonstrates how to back up all files in the directory **/u/usrtools/** that have not been accessed in the last 100 days:

```
pax -wzvf backup.pax.Z    $( find /u/usrtools -type f -atime +100 )
```

Note: The **tar** equivalent for the **pax** portion of the above commands is:

```
tar -czUvf backup.pax.Z
```

Listing process IDs of processes with open files

It is often helpful to know what processes have open files. This information can be provided with the **fuser** utility.

The **fuser** utility lists the process IDs of all processes on the local system that have one or more named files open.

The syntax of the command is as follows:

```
fuser [-cfku] file
```

file is the pathname of the file for which information is to be returned, or, if the —c option is used, the pathname of a file on the file system for which information is to be reported.

Options

—c Reports on all open files within the file system that the specified file is a member of.

—f Reports on only the named files. This is the default for this command.

—k Sends the SIGKILL signal to each local process. Note that only a superuser can terminate a process that belongs to another user.

—u The user name associated with each process ID is written to standard error.

Chapter 17. Handling Security for Your Files

Each user has user ID (UID) and group ID (GID) numbers that are set when the user is defined to the system. A user always belongs to at least one group—for example, a department—and each group that uses the system is assigned a GID. The system uses the UID and GID to identify the files and processes that a user may use. When you create a directory or a file, it is automatically associated with your UID, and its GID is set to the owning GID for the *parent directory* (the directory it is in).

There are three classes of users whose access you can control:

- Owner (the owner of the file or directory, whose UID matches the UID for the file)
- Group (a member of the group whose GID matches the GID for the file)
- Other (anyone else)

You control access to a file and directory *that you own* through its permission bits. (Taken together, the permission bits are often called the *mode*.)

In this chapter, we discuss:

- Default permissions set by the system
- Changing permissions for files and directories
- Using the sticky bit on a directory to control file access
- Auditing file access
- Displaying file and directory permissions
- Setting the file mode creation mask for programs
- Changing the Owner ID or Group ID associated with a file
- Temporarily changing the user ID or group ID during execution

Default Permissions Set by the System

When you first create a file or directory, the system sets default *read*, *write*, and *execute* (`rwx`) permissions. The meanings of the three permissions differ somewhat for a file and a directory:

Permission	Notation	Meaning
read	**r**	**Directory:** Permission to read, but not search, contents. **File:** Permission to read or print contents. To run a shell script, you need both read and execute (discussed below) permission.
write	**w**	**Directory:** Permission to change the directory, adding or deleting members. **File:** Permission to change the file, adding or deleting data
execute	**x**	**Directory:** Permission to search a directory. Usually **r** and **x** are used together. **File:** Permission to run a file—that is, enter it as a command. Typically this permission is used for shell scripts and for files containing executable programs. (To run a shell script, you need read and execute permission.)

The following table shows the default permissions set by the system:

Using	To Create a	Default Permissions
mkdir shell command	Directory	owner=rwx group=rwx other=rwx In octal form: 777
MKDIR TSO command	Directory	owner=rwx group=r-x other=r-x In octal form: 755
JCL with no PATHMODE specified	Directory or file	owner=--- group=--- other=--- In octal form: 000
ISPF editor, OEDIT command, oedit command	File	owner=rwx group=--- other=--- In octal form: 700
vi editor	File	owner=rw- group=rw- other=rw- In octal form: 666
ed editor	File	owner=rw- group=rw- other=rw- In octal form: 666
Redirection (>)	File	owner=rw- group=rw- other=rw- In octal form: 666
cp command	File	Sets the output file permissions to the input file permissions.
OCOPY command	File	Permission bits for a new file are specified with the ALLOCATE command, using the PATHMODE keyword, prior to entering the OCOPY command. If the PATHMODE keyword is omitted, the default is: owner=--- group=--- other=--- In octal form: 000

Using	To Create a	Default Permissions
OPUT or OPUTX command	File	For a text file: owner=rw- group=--- other=--- In octal form: 600 For a binary file: owner=rwx group=--- other=--- In octal form: 700

For more information on octal numbers, see “Using Octal Numbers to Specify Permissions” on page 238.

Changing Permissions for Files and Directories

You can use the **chmod** command to set or change permissions for your files and directories. To change permissions, you must be the owner or a superuser. (If you are uncertain about ownership, use the **ls -l** command and look for your TSO/E user ID.)

You can specify the **chmod** command like this:

chmod *mode pathname*

You can specify the mode in symbolic form or as an octal value. For more information on the **chmod** command, see *OS/390 UNIX System Services Command Reference*.

Using a Symbolic Mode to Specify Permissions

A symbolic mode has the form:

[who] *op permission* [*op permission* ...]

The *who* value is optional; it can be any combination of the following:

u Sets owner (user) permissions.
g Sets group permissions.
o Sets other permissions.
a Sets all permissions; this is the default.

The *op* part of a symbolic mode is an operator that tells **chmod** to turn the permissions on or off. The possible values are:

+ Turns on a permission.
– Turns off a permission.
= Turns on the specified permissions and turns off all others.

To set the *permission* part of a symbolic mode, you can specify any combination of the following permissions in any order:

r Read permission.

s This stands for *set-user-ID-on-execution* or *set-group-ID-on-execution* permission. See “Temporarily Changing the User ID or Group ID during Execution” on page 242 for more information.

t This sets the *sticky bit* on, for a file or directory.

Directory: The sticky bit is set on for a directory so that a user cannot remove or rename a file in the directory unless one or more of these conditions is true:
- The user owns the file.
- The user owns the directory.
- The user has superuser authority.

File: The sticky bit is set for frequently used programs in the file system, to reduce I/O and improve performance. When the bit is set on, OS/390 UNIX System Services searches for the program in the user's STEPLIB, the link pack area, or the link list concatenation. For information on copying a load module from the file system into a data set, see "Copying an Executable Module from the File System" on page 294. See *OS/390 UNIX System Services Planning* for information on using the sticky bit with daemons.

w Write permission. If this is off, you cannot write to the file.

x Execute permission. If this is off, you cannot execute the file.

X Search permission for a directory; or execute permission for a file only when the current mode has at least one of the execute bits set.

For example, to turn on read, write, and execute permissions, and turn off the set-user-ID and sticky bit attributes for a file, enter the command:

```
chmod a=rwx file
```

The user can specify multiple symbolic modes if you separate them with commas.

Using Octal Numbers to Specify Permissions

Typically, octal permissions are specified with three or four numbers, in these positions:

```
1234
```

Each position indicates a different type of access:

- In position 1 are the bits that set permission for set-user-ID on access, set-group-ID on access, or the *sticky bit*. Specifying this position is optional.
- In position 2 are the bits that set permissions for the owner of the file. Specifying this position is required.
- In position 3 are the bits that set permissions for the group that the owner belongs to. Specifying this position is required.
- In position 4 are the bits that set permissions for others. Specifying this position is required.

Position 1

Specifying the bits in position 1 is optional. For position 1, you can specify these octal numbers:

0 Off
1 Sticky bit on.
2 Set-group-ID-on execution
3 Set-group-ID-on execution and set the sticky bit on.
4 Set-user-ID on execution
5 Set-user-ID on execution and set the sticky bit on.
6 Set-user-ID and set-group-ID on execution
7 Set-user-ID and set-group-ID on execution and set the sticky bit on.

Positions 2, 3, and 4

Specifying these bits is required. For each type of access—owner, group, and other—there is a corresponding octal number:

0 No access (`---`)
1 Execute-only access (`--x`)
2 Write-only access (`-w-`)
3 Write and execute access (`-wx`)
4 Read-only access (`r--`)
5 Read and execute access (`r-x`)
6 Read and write access (`rw-`)
7 Read, write, and execute access (`rwx`)

To specify permissions for a file or directory, you use at least a *three-digit* octal number, omitting the digit in the first position. When you specify three digits instead of four, the first digit describes owner permissions, the second digit describes group permissions, and the third digit describes permissions for all others.

If you are not setting the first octal digit, you can just specify 3 digits instead of 4. When the first digit is not set, some typical 3-digit permissions are specified in octal this way:

Table 11. Three-Digit Permissions Specified in Octal

Octal Number		Meaning
666	6 6 6 rw- rw- rw-	owner (rw-) group (rw-) other (rw-)
700	7 0 0 rwx --- ---	owner (rwx) group (---) other (---)
755	7 5 5 rwx r-x r-x	owner (rwx) group (r-x) other (r-x)
777	7 7 7 rwx rwx rwx	owner (rwx) group (rwx) other (rwx)

Using the Sticky Bit on a Directory to Control File Access

Using the **mkdir**, MKDIR, or **chmod** command, you can set the sticky bit on a directory to control permission to remove or rename files or subdirectories in the directory. When the bit is set, a user can remove or rename a file or remove a subdirectory only if one of these is true:

- The user owns the file or subdirectory.
- The user owns the directory.
- The user has superuser authority.

If you use the **rmdir**, **rename**, **rm**, or **mv** utility to work with a file, and you receive a message that you are attempting an “operation not permitted”, check to see if the sticky bit is set for the directory the file resides in.

Auditing File Access

Using the **chaudit** command, you can specify which types of file access are audited by RACF. RACF writes the audit information to system management facilities (SMF) record 80.

Only a file owner or a security auditor can specify if auditing is turned on or off, and when audit records should be written for a directory or a file: for successful accesses, failed accesses, or for all accesses.

You can specify audits for read, write, and search or execute attempts. For each of these, you can specify audits for successful access, failed access, or both. You can also set the audit flags off, so audits are not performed.

The default audit bits are set at file creation:

- The user-requested-audit flags are set to audit failed attempts to read, write, or execute. Only the file owner or a superuser can specify user audit options.
- The auditor-requested-audit flags are set off (no auditing). To specify auditor audit options, one must have security auditor authority.

See *OS/390 UNIX System Services Command Reference* for a description of the **chaudit** command. See *OS/390 UNIX System Services Planning* for a description of how a superuser or security auditor would use the **chaudit** command.

Displaying File and Directory Permissions

To display the permissions for the files and directories in your working directory, use **ls -W**. (The **ls -l** command displays all the access permissions but does not display the audit permissions.) The display format is:

```
drwxr-x---  fff---    2  ELVIS  64MB   96 Jun 15 10:34 statrp
-rwx------  fff---    1  ELVIS  64MB  107 Jul 10 07:45 jun93
-rwx------  fff---    1  ELVIS  64MB   80 Aug 09 13:15 jul93
-rwx------  fff---    1  ELVIS  64MB  150 Sep 15 10:45 aug93
drwxr-xr-x  fff---    2  ELVIS  64MB   96 Jun 17 09:05 dbappl
-rwxr-x---  fff---    1  ELVIS  64MB  150 Jun 17 10:15 txn1
```

First field: A string of 10 characters. The first character indicates the file type. The next 9 characters are the permissions. For example:

```
-rwxr-xr-x
```

View them this way:

```
- rwx r-x r-x
```

– The first character indicates whether this is a file or directory.
  - for a regular file (binary or text)
  c for a character special file
  d for a directory
  e for an external link
  l for a symbolic link
  p for a named pipe (FIFO special file)

  In the example, **-** indicates a regular file.

– The first set of 3 characters show the owner's permissions. In this example, the owner has read, write, and execute permission (`rwx`).

– The second set of 3 characters show the group permissions. In this example, the group to which the user belongs has read and execute permission (`r-x`).

– The third set of 3 characters show the “other” permissions. In this example, any other user can read the file and execute it (`r-x`). If the sticky bit is on, you see a `T` or `t` in the final field (`--T` or `--t`).

Second field: The audit settings. These 6 characters are actually two groups of 3 characters. The first group of 3 describes the audit settings requested by a user; the second group describes audit settings requested by a security auditor. The characters can be:

s to audit successful access attempts

f to audit failed access attempts

a to audit all accesses

- for no audit

In the example, `fff---`,

fff means *failed* read, write, and execute or search attempts to access the file are audited by the user.

--- means read, write, and execute or search attempts to access the file are not audited by the security auditor.

Third field: The number of links to the file or directory.

Fourth field: The owner’s login name (TSO/E user ID).

Note: When files owned by user ID 0 (UID=0) are transferred from any UNIX-type system across a NFS connection to another UNIX-type system, the user ID changes to -2 (UID=-2). On an OS/390 UNIX system, -2 is not a valid user ID, therefore **ls** displays UID 4294967294 (the unsigned equivalent of -2).

Fifth field: The name of the group associated with the file or directory.

Sixth field: The size of the file, expressed in bytes.

Seventh field: A date and time. For a file, this is the time the file was last changed; for a directory, it is the last time a file was created or deleted in the directory.

Eighth field: The name of the file or directory. If the file is a symbolic link, that also is indicated. See the additional information for the filename **lnk** in this example:

```
l---------   1 ELVIS    SYS1          8 May 21 15:30 lnk -> /tmp/ehk
$
```

Setting the File Mode Creation Mask

When a file is created, it is assigned initial access permissions. If you want to control the permissions that a program can set when it creates a file or directory, you can set a *file mode creation mask* using the **umask** command.

The user can set this file mode creation mask for one shell session by entering the **umask** command interactively, or you can make the umask command part of your login. When you set the mask, you are setting limits on allowable permissions: You are implicitly specifying which permissions are *not* to be set, even though the calling program may allow those permissions. When a file or directory is created, the permissions set by the program are adjusted by the **umask** value: The final permissions set are the program’s permissions minus what the umask values restrict.

To use the **umask** command for a single session, enter:

umask *mode*

and specify the mode in either of the formats used by **chmod**: symbolic (`rwx`) or octal values. The symbolic form expresses what can be set, what is *allowed*, while octal values express what cannot be set, what is *disallowed*. For example, both of these commands set the same umask:

```
umask a=rx
umask 222
```

To display the mask,

- If you just enter **umask**, you see the mode displayed in octal values, indicating what *cannot* be set.
- If you enter **umask -S**, you see the mode displayed in symbolic form, indicating what *can* be set.

The shell's initial setting of the mask is 000, which means that read, write, and execute permission can be set on for everyone. But the systemwide profiles provided with the product set the mask to 022.

Changing the Owner ID or Group ID Associated with a File

The user might need to change the UID or GID for a file. To protect the data in a file from unauthorized users, the system controls who can change the file access:

- To change the owner (UID) of a file, the superuser can enter a **chown** command.
- To change the group (GID) of a file, the superuser or the file owner can enter a **chgrp** command, specifying either a RACF group name or a GID. The file owner must have the new group as his group or one of his supplementary groups.

Superuser tasks are discussed in *OS/390 UNIX System Services Planning*.

Temporarily Changing the User ID or Group ID during Execution

An executable file can have an additional attribute, which is displayed in the execute position (**x**) when you issue **ls -l**. This permission setting is used to allow a program temporary access to files that are not normally accessible to other users. An **s** or **S** can appear in the execute permission position; this permission bit sets the effective user ID or group ID of the user process executing a program to that of the file whenever the file is run. The setuid and setgid bits are only honored for executable files containing load modules. These bits are not honored for shell script and REXX execs that reside in the file system.

s In the owner permissions section, this indicates that both the set-user-ID (S_ISUID) bit is set *and* execute (search) permission is set.

In the group permissions section, this indicates that both the set-group-ID (S_ISGID) bit is set *and* execute (search) permission is set.

S In the owner permissions section, this indicates the set-user-ID (S_ISUID) bit is set, but the execute (search) bit is not.

In the group permissions section, this indicates the set-group-ID (S_ISGID) bit is set, but the execute (search) bit is not.

A good example of this behavior is the **mailx** utility. A user sending mail to another user on the same system is actually appending the mail to the recipient's mail file, even though the sender does not have the appropriate permissions to do this—the mail program does.

Displaying Extended Attributes

-E is an option on the **ls** shell command that will display extended attributes. For more information on this option, refer to “Executable Modules in the File System” on page 198.

Chapter 18. Editing Files

When logged into the shell, you have a choice of editors to use to create and change files, depending on which interface you are using:

OMVS Terminal Interface:

- The full-screen ISPF editor that you can invoke using the OEDIT or **oedit** command.
- The **ed** editor, a line editor
- The **sed** stream editor, a noninteractive editor. It is intended for *systematic* editing; you invoke the editor with a file of editing commands and a target data file and it produces an edited target file, with no user interaction.

Asynchronous Terminal Interface:

- The **vi** editor, an interactive editor

If you are using NFS from your workstation, you can directly edit HFS files with your workstation editor of choice.

Using ISPF to Edit an HFS File

Using ISPF:

ISPF Edit provides a full-screen editor you can use to create and edit HFS files. You can access ISPF Edit in several ways:

- Using the **oedit** shell command
- Using the TSO/E OEDIT command at the TSO/E `READY` prompt or from the shell command line
- From the ISPF menu (if a menu option is installed)
- From the ISPF shell (accessed using the TSO/E ISHELL command)

Note: If you know you will be using OEDIT or OBROWSE during a shell session, make your initial invocation of the shell from ISPF. If you enter the OMVS command from ISPF, you can subsequently access OEDIT and OBROWSE more quickly than if you had entered the OMVS command from TSO/E.

Using ISPF Edit, you can edit only regular files (not special files). You need read and write permission for the file and search permission for any intermediate directories.

When you are working in MVS (TSO/E or ISPF), your home directory is the default working directory.

When you create a new file, you must have the appropriate permissions to add a new file to the parent directory. When a file is created using ISPF Edit, its default permissions are

owner = `rwx`
group = `---`
other = `---`

The octal number is 700.

ISPF Edit allows only one edit session at a time per file. It reads the entire file when the edit session begins. At the end of the session, it replaces the original file with the edited file.

During an ISPF Edit session, you can use these types of commands:

Scrolling commands You can use commands to scroll the data up, down, left, or right.

Line commands You perform line editing by entering a *line command* directly on the line number of the affected line. For example, to delete a line, you enter `D` on the line number; to repeat a line, you enter `R` on the line number. You can enter line commands for several lines at the same time.

Primary commands To perform general editing tasks, you enter *primary commands* at the command line on the panel. For example, you can use the FIND command to scan data for a specific character string. If you entered:

```
FIND printf(
```

on the command line, your cursor moves to the next occurrence of **printf(**. Likewise, you can enter the CHANGE command to make global changes within a file. For example:

```
CHANGE CRTL C-RTL ALL
```

changes all instances of `CRTL` to `C-RTL`.

External data commands While you are editing one file, you can use external data commands to work with another file, a sequential data set, or a member of a partitioned data set or PDSE—moving data to or from the file you are editing. ISPF Edit provides five external data commands: COPY, MOVE, REPLACE, CREATE, and EDIT. See “Working with Another File or a Data Set While Editing a File” on page 250.

To end an edit session:
- Saving all changes, enter the END command or press <F3>.
- Without saving any changes, enter the CANCEL command.

When you end the edit session, you go back to where you were when you began it: on the entry panel, on an ISPF command line, at the TSO/E `READY` prompt, or at the shell prompt.

All You Ever Wanted to Know about ISPF Edit

The discussion in this chapter is an introduction to ISPF Edit. For detailed information about ISPF Edit, including the commands just mentioned, use the online help facility or refer to *ISPF/PDF Edit and Edit Macros*.

Support for Doublebyte Characters

The ISPF editor works with doublebyte characters. Mixed mode is the assumed operation mode. You cannot specify a data set format, since this requires fixed-length records and lines in the hierarchical file system are always treated as variable-length records.

Code Page Conversion

When you edit an HFS file using ISPF Edit, two code pages may be at work and *there is no conversion between them*. If you have not customized your keyboard,

any left or right square bracket you type in ISPF may be stored as characters that will not be properly interpreted by the C compiler, shell, or utilities. For a discussion of code page conversion, see “Appendix E. Code Page Conversion when the Shell and MVS Have Different Locales” on page 345.

Typing Tabs in ISPF

Writing makefiles for the **make** utility requires the use of a <tab> character. **awk** programs can also use tabs.

If you are using an OS/390 UNIX editor, you can type a tab character as an <EscChar-I> sequence. When you press <Enter>, a blank space is displayed.

If you are using ISPF Edit, you cannot type a tab character (ISPF handles only displayable characters). Instead, you can:

1. Select a substitute for a tab character, for example, the character @.
2. At the beginning of each line of the makefile, type an @ instead of a tab character.
3. When you have finished editing the file, on the command line enter:

   ```
   change @ X'05' all 1
   ```

   This converts every @ in column 1 to the hexadecimal character 05, which is a tab. The `1` at the end of the command is important because it restricts the change to the @ in column 1—leaving unchanged any other @s you might have in the file. (Alternatively, you can use a tab substitute character that you know is not used in the file, such as ¢.)
4. In ISPF Edit, the X'05' now displays as a blank space, which is protected; you cannot type over it.

The foregoing is just one of several methods you can use to edit hex data.

If you use ISPF to edit or browse an existing file that has tabs in it:

- In browse mode, the X'05' (tab) displays as a period (`.`) by default.
- In edit mode, the X'05' displays as a blank space. When you edit the file, ISPF displays a message that the file contains “unprintables” (in this case, the tab characters) and tells you how to use the FIND command to locate them. You can change the tabs back to @ by entering the following command:

  ```
  change X'05' @ all 1
  ```

Preserving Trailing Blanks in Files

ISPF Edit removes trailing blanks from lines in files. If this is undesirable, use the **sed** stream editor instead.

Working with Lowercase or Mixed-Case Files

Suppose you are creating a new file and the ISPF Edit default action set on your system is to convert to uppercase the characters you type in a file. If you want the data in this file to be in lowercase or mixed-case letters, enter `caps off` on the command line. This prevents ISPF from converting the letters you type into uppercase. After you enter `caps off`, it remains in your profile

For editing an existing file,

- If it has any lowercase letters, ISPF sets CAPS OFF mode.
- If it has only uppercase letters, ISPF sets CAPS ON mode.

If you are working on a file and realize that you have been typing in uppercase when you really wanted lowercase, you can change the contents of the file to all lowercase. Type this on the command line:

```
c all p'>' p'<'
```

Editing a File with Long Records

With ISPF Version 4 and ISPF Edit, you can edit files that have records longer than 255 characters (bytes). The width of the editing session is calculated from the length of the longest record with 25 percent added to allow for some expansion.

Accessing a File to Edit

1. To use ISPF to edit a regular file (not a special file), you can do one of the following:
   - From the ISPF shell (ISHELL), select `Edit(E)` from the File Pulldown.
   - Select an option for editing HFS files on the ISPF menu, if such an option is available. This displays the ISPF Edit Entry panel shown in Figure 31 on page 249.
   - Use the **oedit** command. You can enter the command followed by an absolute or relative pathname. If you type **oedit** with no pathname, this displays the ISPF Edit Entry panel, shown in Figure 31 on page 249.
   - Use the TSO/E OEDIT command. You can enter the command in several ways:
     – OEDIT followed by the absolute pathname for the file. This command displays the file, and you can begin editing.
     – OEDIT followed by the absolute pathname for a file that does not exist. This creates a new file.
     – OEDIT with no filename. This displays the ISPF Edit Entry panel, shown in Figure 31 on page 249.

     Note: If you are working in the shell and you invoke OEDIT with a relative filename, OEDIT searches for the file in your home directory. This is because the working directory of the TSO/E session is typically your home directory.

     Attention! ... If You Enter the OEDIT Command in ISPF
     To get case-sensitive processing of the filename when you enter `oedit filename` or `obrowse filename` in ISPF, enter the command on a command line that supports mixed-case processing—for example, the "Command Processor" panel (usually ISPF option 6). Some ISPF option panels convert the command and filename to uppercase before processing them.
2. Complete the Edit Entry panel, if it is displayed, and press <Enter>.

```
---------------------- EDIT - ENTRY PANEL ---------------------------
Command ===>

Directory      ===>


Filename       ===>

Profile name   ===>

Initial macro  ===>
```

Figure 31. ISPF Edit Entry Panel for an HFS File

These are the fields on the panel:

Directory
Specify the directory name in uppercase, lowercase, or mixed-case characters. The directory name can be up to 200 characters long. Do not use a space or null character in the name.

When you are working in MVS (TSO/E or ISPF), by default your working directory is your home directory.

Filename
Specify the name of a new or existing file, up to 44 characters long. We suggest you use the characters in the *POSIX portable filename character set*: uppercase or lowercase characters A to Z, numbers `0` to 9, an underscore, hyphen, or period. Do not use a slash in the filename.

A new file is created with the file mode set to 700 (read, write, and execute for the owner). You must have the appropriate permissions to add a new file to the parent directory.

Profile name
This is the name of an edit profile you can use to override the default edit profile, which is used if you leave the field blank. The edit profile controls edit modes (CAPS, NUMBER, and so on) and special definition lines (MASK, TABS, and so on). Specifying a profile name is standard ISPF/PDF practice; you can read more about it in *ISPF/PDF Edit and Edit Macros*.

If you specify an edit profile to use when editing HFS files, be sure that the profile has the `NUMBER OFF` option. This prevents the editor from putting line numbers in columns 1–8.

How OEDIT Selects the Default Profile

If the filename does not contain a period (`.`) or if there are no characters after the period, the HFSPROF profile is used.

If the name does have a period with characters after it, the default profile name consists of the characters after the period. For example, for the file named **/u/project/prog.c**, the profile name is **c**.

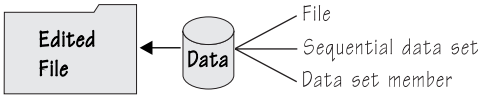
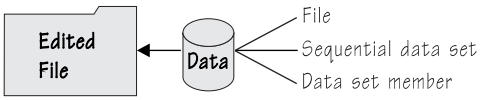
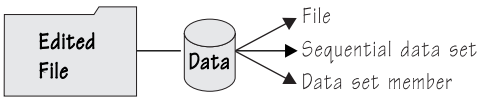
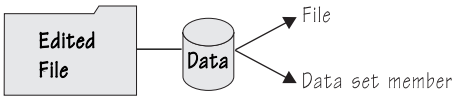
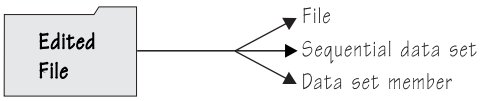
Initial macro
This is an edit macro that runs before the data is displayed. If you do not specify a macro name, none is used. Specifying an initial macro is standard ISPF/PDF practice.

3. If you are creating a new file and want to enter the data in lowercase or mixed-case letters, enter `caps off` on the command line. This prevents ISPF from folding the letters you type into uppercase. After you enter `caps off`, it remains in your profile.

Working with Another File or a Data Set While Editing a File

While editing an HFS file, you can type an external data command on the command line to work with an additional file, a partitioned data set member, or a sequential data set. You can enter the following external data commands:

Table 12. ISPF Edit: External Data Commands

COPY		Copies another file, data set, or member of a partitioned data set or PDSE into the file you are editing.
MOVE		Moves another file, data set, or member of a partitioned data set or PDSE into the file you are editing and deletes the source file, data set, or member.
REPLACE		Replaces the contents of a file, data set, or member of a partitioned data set or PDSE with lines from the file you are editing.
CREATE		Creates a file or member of a partitioned data set or PDSE with lines from the file you are editing.
EDIT		Edits another file, data set, or member of a partitioned data set or PDSE during your current edit session.

Using Edit Macros

If you use edit macros that depend on data set attributes or assume that you are editing a data set or partitioned data set member, you may need to tailor the macros to work with an HFS file. The following attributes may not have meaning for files:

BLKSIZE
DATASET

LRECL
MEMBER
RECFM
STATS

For more information, see *ISPF/PDF Edit and Edit Macros*.

Copying into a File

Use the COPY command to copy an entire file or one or more lines from a sequential data set or a member of a partitioned data set (PDS) into the file you are editing now.

1. Specify `copy` on the command line.
2. Specify the line command A (after) or B (before) on a line number to indicate where the data is to be copied.
3. Press <Enter>. The Edit Copy panel is displayed; see Figure 32.

```
---------------------- EDIT - COPY PANEL ----------------------------
Command ===>

Directory      ===>


Filename       ===>

Use data set   ===>       Y to select data set panel
```

Figure 32. ISPF Edit Copy Panel for an HFS File

4. Specify the data to be copied on the Edit Copy panel. To complete this panel, indicate if the data is in a file (in which case, you supply its directory name and filename) or a data set:

   Directory
   The directory name you specify on this panel must be an existing directory. Type the name in uppercase, lowercase, or mixed-case characters. The name can be up to 200 characters long, without a space or null character in it. Leave this blank if you want to work with a partitioned data set or PDSE member.

   Filename
   The name can be up to 44 characters long. Leave this blank if you want to work with a partitioned data set or PDSE member.

   Use data set
   If you want to work with a data set or partitioned data set or PDSE member, type `Y`.

5. Press <Enter>.
   - If you are copying from an HFS file, the copy is completed.
   - If you are copying from a data set or partitioned data set or PDSE member, the ISPF Edit Copy panel for a data set is displayed. You supply information about the data set and you can specify the first and last lines of the text to be copied.

Moving Data into a File

Use the MOVE command to move a file, a sequential data set, or a member of a partitioned data set into the file you are editing. After the data is moved into the file, the source file or data set is deleted.

1. Specify `move` on the command line.
2. If the file that you are editing is not empty, specify the line command A (after) or B (before) on a line number to indicate where the data is to be inserted.
3. Press <Enter>. The Edit Move panel is displayed. It looks similar to Figure 32 on page 251.
4. Specify the data to be moved on the Edit Move panel. To complete this panel, indicate if the data is in a file (in which case, you supply its directory name and filename) or a data set:

   Directory
   The directory name you specify on this panel must be an existing directory. Type the name in uppercase, lowercase, or mixed-case characters. The name can be up to 200 characters long, without a space or null character in it. Leave this blank if you want to work with a partitioned data set or PDSE member.

   Filename
   The name can be up to 44 characters long. Leave this blank if you want to work with a partitioned data set or PDSE member.

   Use data set
   If you want to work with a data set or partitioned data set or PDSE member, type `Y`.
5. Press <Enter>.
   - If you are moving data from an HFS file, the move is completed.
   - If you are moving data from a data set or partitioned data set or PDSE member, the Edit Move panel for a data set is displayed. You supply information about the data set to be moved.

Replacing a File or Data Set with Data from a File

Use the REPLACE command to replace the contents of a file, a sequential data set, or a member of a partitioned data set or PDSE with one or more lines from the file you are editing.

1. Type `replace` on the command line. To identify the lines to be put into the file or data set, use any of these line commands:
   - Specify C (copy) or M (move) on a line number for a single line of data.
   - Specify CC or MM on the beginning and ending line numbers for a block of data.
   - Specify C*nnnn* or M*nnnn* on a line number, where the *nnnn* refers to the number of subsequent lines to be copied or moved.

   If you move lines, they are deleted in the source file.
2. Press <Enter>. The Edit Replace panel is displayed. It looks similar to Figure 32 on page 251.
3. Specify what you are replacing on the Edit Replace panel. To complete this panel, indicate if you want to move data into a file (in which case, you supply its directory name and filename) or a data set:

   Directory
   The directory name you specify on this panel must be an existing directory. Type the name in uppercase, lowercase, or mixed-case

characters. The name can be up to 200 characters long, without a space or null character in it. Leave this blank if you want to work with a partitioned data set or PDSE member.

Filename
The name can be up to 44 characters long. Leave this blank if you want to work with a partitioned data set or PDSE member.

Use data set
If you want to move data into a data set or partitioned data set or PDSE member, type `Y`.

4. Press <Enter>.
   - If you are replacing an HFS file, the data is replaced.
   - If you are replacing a data set or partitioned data set or PDSE member, the Edit Replace panel for a data set is displayed. You supply information about the data set to be replaced.

Creating Another File or Data Set with Data from a File

Use the CREATE command to create a new file or member of a partitioned data set or PDSE, using data from a file you are editing. (You cannot create a sequential data set.)

1. Type `create` on the command line. To identify the lines to be put into the file or data set being created, use these line commands:
   - Specify C (copy) or M (move) on a line number for a single line of data.
   - Specify CC or MM on the beginning and ending line numbers for a block of data.
   - Specify C*nnnn* or M*nnnn* on a line number, where the *nnnn* refers to the number of subsequent lines to be copied or moved.

   If you move lines, they are deleted in the source file.
2. Press <Enter>. The Edit Create panel is displayed. It looks similar to Figure 32 on page 251.
3. Specify what is to be created on the Edit Create panel. To complete this panel, indicate if you are creating a file (in which case, you supply its directory name and filename) or a data set:

   Directory
   The directory name you specify on this panel must be an existing directory. Type the name in uppercase, lowercase, or mixed-case characters. The name can be up to 200 characters long, without a space or null character in it. Leave this blank if you want to work with a partitioned data set or PDSE member.

   Filename
   The name can be up to 44 characters long. Leave this blank if you want to work with a partitioned data set or PDSE member.

   Use data set
   If you want to work with a data set or partitioned data set or PDSE member, type `Y`.

   Type `Y` for “Use data set” if you want to create a file or a partitioned data set or PDSE member with data from the file you are editing. If your target is a file, supply the directory name and filename; see the instructions on page 249.
4. Press <Enter>.
   - If you are creating an HFS file, the file is created.

- If you are creating a partitioned data set or PDSE member, the Edit Create panel for a data set is displayed. You supply information about the partitioned data set or PDSE member to be created.

  Note: CREATE adds a member to a data set only if a member of the same name does not exist.

Editing Another File or Data Set during an Edit Session

Use the EDIT command during your current editing session to edit another file, a sequential data set, or a member of a partitioned data set or PDSE. If the file does not exist, it is created.

1. Type `edit` on the command line and press <Enter>. The Edit Edit panel is displayed. It looks similar to Figure 32 on page 251.
2. Specify the data to be edited on the Edit Edit panel. To complete this panel, indicate if you want to edit a file (in which case, you supply its directory name and filename) or a data set:

   Directory
   The directory name you specify on this panel must be an existing directory. Type the name in uppercase, lowercase, or mixed-case characters. The name can be up to 200 characters long, without a space or null character in it. Leave this blank if you want to work with a partitioned data set or PDSE member.

   Filename
   The name can be up to 44 characters long. Leave this blank if you want to work with a partitioned data set or PDSE member.

   Use data set
   If you want to edit a data set or partitioned data set or PDSE member, type `Y`.
3. Press <Enter>.
   - If you are editing an HFS file, the file is displayed.
   - If you are editing a sequential data set or partitioned data set or PDSE member, the Edit Command Entry panel is displayed. You supply information about the data set to be edited.

Edit Recovery

The edit profile you use contains the setting of recovery mode. You can turn it on by entering the primary command RECOVERY ON on the command line when you are editing a file.

When RECOVERY mode is on and a system problem occurs during an edit session, all changes you made are saved. For example, say a communication problem dropped your line, or there was a LAN failure, or your workstation was powered off accidentally—in all these cases, recovery is automatic. With edit recovery, you can continue editing from the last interaction.

The next time you use ISPF Edit, the Edit Recovery panel is displayed. The pathname of the recovered file is displayed on the panel. You can recover the file, defer the recovery, or cancel the recovery, as indicated on the panel in Figure 33 on page 255.

```
------------------------ EDIT - RECOVERY -----------------------------
Command ==>

                 *******************************
                 *    EDIT AUTOMATIC RECOVERY  *
                 *******************************

The following file was being edited when a system failure or task abend
occurred:

Filename:




Instructions:
   Press ENTER key to continue editing the file, or
   Enter END command to return to the primary option menu, or
   Enter DEFER command to defer recovery of the specified file, or
   Enter CANCEL command to cancel recovery of the file.
```

Figure 33. The Edit Recovery Panel for an HFS File

To recover a file, the next time you use ISPF Edit, ensure that you are working in the same directory that you used when you originally edited the file. If you specify a relative pathname for a file when you start to edit it (for example, OEDIT **prog/appl.c**), that relative pathname is used in edit recovery.

Using the vi Screen Editor

The **vi** editor is available if you login to the shell using **rlogin** or **telnet**. It is not available if you login using the OMVS command. The **vi** editor is a full-featured text editor with the following major features:

- Full-screen editing and scrolling capability
- Separate text entry and edit modes
- Global substitution and complex editing commands using the underlying **ex** commands.

This overview just introduces some fundamentals to help you get started. For more information, see Appendix C and *OS/390 UNIX System Services Command Reference*.

Basic Principles

To begin using **vi**, you type the command:

```
vi filename
```

where *filename* is the name of a file you want to edit. This can be an existing file, or it can be a new file that you want to create.

The **vi** command begins a **vi** session. In a **vi** session, you enter input that creates or changes the contents of the file specified on the command line. **vi** reads and uses the input you type until you quit your **vi** session.

In a **vi** session, you are always in one of two modes:

- In **Insert Mode**, everything you type is taken as text input. **vi** displays text on the screen as you enter it. Eventually, **vi** stores this text in a file.

- In **Command Mode**, **vi** interprets everything you type as a command to change the text in some way. Usually, commands do not appear on the screen—you just see the effects of the command. For example, if you enter the command to delete a line of text, you see the line disappear, but you never see the delete line instruction that you actually typed.

To switch from Insert Mode to Command Mode, simply press the key marked <Esc>. If you are not sure of what mode you are in, press <Esc> several times. This always brings you back to Command Mode.

To delete a character, you must be in Insert Mode. Pressing <Backspace> deletes the last character you typed; pressing <Backspace> twice deletes the last two characters, and so on. **vi** usually does not immediately delete these characters on the screen—it just backs up the cursor so that anything you enter is typed over the characters that were there. When you leave Insert Mode, **vi** adjusts the screen to remove any characters that were deleted by <Backspace> and not over-typed.

To quit a **vi** session, do one of these:
- **:wq** to save your changes and quit **vi**
- **:q!** to quit without saving your work

A Simple vi Session

This section shows you how to edit a simple text file. Try it to get the feel of using **vi**. You can edit the text file:

```
vi1.txt
```

which is supplied as part of the OS/390 shell. It is in the directory **/samples**. To do this, copy this file to current working directory:

```
cp /samples/vi1.txt vitest
```

Now, begin your **vi** session by typing:

```
vi vitest
```

vi clears the screen, then displays the contents of the file. At the bottom of the screen, **vi** also displays:

```
"vitest" 30 lines, 668 characters
```

This tells the name of the file being edited and how big it is.

The cursor is positioned at the beginning of the file. These keys let you position the cursor anywhere on any line in the file:

Action Key

Move the cursor down a line.
Press **j** or ↓ (the Down arrow key)

Move the cursor up a line.
Press **k** or ↑ (the Up arrow key)

Move the cursor left along a line.
Press **h** or ← (the Left arrow key)

Move the cursor right along a line.
Press **l** or → (the Right arrow key)

Note: The arrow keys do not work on all terminals

To experiment a bit more, move the cursor to the beginning of the first line in the file, then press 5 followed by →. You do not see the 5 displayed anywhere—but when you press →, you see the cursor move five characters to the right. As a general rule, when you type a number followed by an action, **vi** repeats the action that number of times.

By the way, ask yourself if you are in Insert Mode or Command Mode. You must be in Command Mode because the characters you type (for example, the 5) do not appear on the screen. When you start a **vi** session, you always begin in Command Mode.

Adding Text

The simplest action you can perform is adding text to what is already on the screen. Move the cursor to the blank line following:

```
And frightened Miss Muffet away.
```

The cursor should be at the first position in the blank line. Now type **a**. Since you are in Command Mode, this is taken to be a command, not text. The **a** command tells **vi** to begin adding to the text that already appears on the screen. If you now type:

```
Little Boy Blue
```

you can see the characters appear on the line. The **a** command switches from Command Mode to Insert Mode. You can now see what you are typing.

Press <Enter> at the end of the line. The bottom part of the screen moves down to make a new blank line after the line you were typing. Keep typing more lines:

```
Come blow your horn
The sheep's in the meadow,
The cow's in the corn.
```

You see that the bottom part of the screen keeps moving down to make more room for what you are typing. After the **a** command, the text that you type is added into the middle of existing text.

When you have typed the last line, press <Enter> to make a new blank line, then press <Esc>. <Esc>switches from Insert Mode back to Command Mode. Now, **vi** interprets what you type as commands again. If you type 4 followed by ↑, the cursor moves up four lines to the beginning of the text you just typed in. The 4 does not appear on the screen when you type it, because command input is not usually displayed.

Move the cursor to the B at the beginning of the word Blue in the text you have just typed. Press **a** to add more text, then type the letter `l`. The `l` is added after the B and the rest of the text on the line shoves over to make room for the new character. This shows that **a** adds text after the current cursor position.

Press <Backspace>. The cursor backs up one space. Press <Esc>to return to Command Mode. The `l` disappears when you leave Insert Mode, and **vi** adjusts the screen to get rid of characters deleted by backspacing.

The Little Boy Blue rhyme that you have just added to the file follows the previous nursery rhyme immediately. The file would look better with a blank line separating the two rhymes. Figure out how to put in this blank line, and do it.

Moving the Cursor Up and Down the Screen

You already know how to move the cursor up and down; however, this can be a slow process if you have a large file that you want to move through quickly. To speed this process up, **vi** offers several commands that can jump the cursor up or down many lines at a time.

In Command Mode, use the following commands:

Command	Moves the Cursor:
H	To the upper left hand corner of the screen. **H** stands for High and it moves the cursor as high on the screen as it can go.
L	To the bottom of the screen. **L** (uppercase) stands for Low.
M	To the middle of the screen. **M** stands for Middle. Experiment with these commands to see how they move the cursor.

Moving Up and Down through a File

While you are editing a file, you can move through it one line at a time, several lines at a time, or screens at a time. You can use these commands to move up and down through a file:

Command	Moves the Cursor:
<Ctrl-D>	Down (or forward) half a screen. The cursor stays where it is -- the text moves underneath it.
<Ctrl-F>	Down (or forward) almost a full screen. This lets you move forward through the file very rapidly.
<Ctrl-U>	Up (or backwards) half a screen.
<Ctrl-B>	Up (or backwards) almost a full screen.

If you move forward far enough through **vitest**, you will see a number of lines that are blank except for a tilde (~) as the first character. These lines are actually beyond the end of the file -- the file ends with the line:

```
And the mome raths outgrabe.
```

vi could just show an empty screen after this last line, but then you would not know if the screen was empty because you had reached the end of the file or if the file just contained a lot of blank lines; therefore, **vi** uses ~ to mark lines that are past the end of the file.

Moving the Cursor on the Line

You can also move the cursor by whole word boundaries, using word-motion commands. Make sure that you are in Command Mode (press <Esc>). **0** and **$** let you move back and forth on a line quickly.

Command	Moves the Cursor:
^ or **0**	To the beginning of the current line (to the first nonblank space). The command **0** is short for **0l**, which moves the cursor to column number 0.
$	To the end of the current line

$ stands for the end of the line in a number of **vi** commands.

Go to the beginning of a line, and press **w**. The cursor jump forward to the beginning of the next word on the line. **w** stands for word and it moves the cursor forward one word. If you keep pressing **w**, the cursor keeps jumping forward. When you jump forward from the last word in the line, you go to the first word in the next line. If precede **w** with a number (as in **5w**), the cursor jumps forward that many words.

Typing **b** works like **w** except that you go back a word instead of forward. If you go back from the first word on a line, you get to the last word on the previous line. If you precede **b** with a number (as in **3b**), the cursor jumps backward that many words.

If the cursor is in the middle of a word, typing **e** moves the cursor to the end of the word. For example, if the cursor is in the middle of the word `slithy`, typing **e** goes to the last letter in the word. If the cursor is already on the last letter of a word, typing **e** moves the cursor to the end of the next word.

To move the cursor between words *including punctuation* (that is, punctuation is considered to be a word), use the following commands:

Command	Moves the Cursor:
e	To the end of current word
w	To the beginning of the next word
b	To the beginning of the previous word

To move the cursor between words *ignoring punctuation* (that is, punctuation is skipped), use the following commands:

Command	Moves the Cursor:
E	To the end of the current word
W	To the beginning of the next word
B	To the beginning of the previous word

Moving to Sentences and Paragraphs

To move between sentences and paragraphs, use the following commands:

Command	Moves the Cursor:
)	To the beginning of the next sentence.
(	To the beginning of the preceding sentence.
}	To the beginning of the next paragraph.
{	To the beginning of the preceding paragraph.

These commands can also be preceded by a number to change the effect of the command. For example, **3)** moves the cursor forward 3 sentences.

Deleting Text

There are several commands that delete text from the screen. All of these begin with the letter **d**. After the **d** comes a letter indicating what you want to delete. Usually this letter is based on one of the cursor movement commands. For example:

Command	Action

d$	deletes text from the cursor's current position to the end of the line.
dd	Deletes the entire line containing the cursor
dL	Deletes text from the cursor's current position to the bottom of the screen.
dw	Deletes text from the cursor's current position to the beginning of the next word.
de	Deletes text from the cursor's current position to the end of a word. If the cursor is in the middle of a word, **de** deletes to the end of the same word; if the cursor is at the end of a word, **de** deletes to the end of the next word.

In the same way, **d** followed by → or ← (**l** or **h**) can delete a single character. Try both instructions and see which character gets deleted.

If you delete something by accident, you can undo the deletion by typing **u** (lowercase). Try this now. Type **dH**. What happens? Now type **u** and see the deleted text return.

A number followed by a delete command repeats the command that number of times. For example:
- **5dw** deletes five words
- **10dd** deletes ten lines.

Changing Text

To change existing text, use the **c** command the same way you use **d**. **c** is a combination of **d** and **a**—it deletes text, then begins to append text to replace what was deleted.

Command	Action
c$	Lets you change everything from the cursor's current position to the end of the line.
cL	Lets you change everything to the end of the page.
cc	Lets you change all of the current line, regardless of the cursor position.

Go to the beginning of the first line of **vitest** and type **c$**. **vi** puts a `$` at the end of the line. The `$` marks the end of the block of text that **vi** intends to change. If you now begin typing something like `The rain in Spain`, you type over the text that was previously on the line. If you keep typing, you eventually type over the `$`. The `$` was never there -- it was just a marker to show the block of text to be replaced.

After a **c** command, the text you type shows up on the screen. This means that **c** puts you in Insert Mode. When you finish typing replacement text, you must press <Esc>to return to Command Mode.

You can enter any amount of text to replace existing text. For example, **c$** only gets rid of part of a line, but you can enter many lines of replacement text.

Undoing a Command

If you make a change and then realize it was in error, you may still be able to correct it.

Command	Action
u	Undoes the last command entered
U	Undoes all changes made to the current line

Saving a File

When you finish editing text, you must save your work in a file. Until you save your work, your text is on the screen but is not recorded in any usable way. When you quit **vi**, your work disappears unless it is saved.

If you started your **vi** session with `vi filename`, it is easy to write the edited text back into the same file. In Command Mode, just type:

```
:w
```

and press <Enter>. When you type the colon, it appears at the very bottom of the screen. The **w** also appears at the bottom of the screen. When you press <Enter>, there is a short pause and then **vi** displays some statistics about the saved text: the name of the file, and the number of lines and characters saved.

If you want to save your changes and quit **vi**, enter:

```
:wq
```

If you want to save your text in a different file, type:

```
:w newfilename
```

and press <Enter>. Again, this appears at the bottom of the screen. After you save your work, you can quit **vi** by typing:

```
:q
```

Normally, **vi** does not let you quit before saving; if you do, you lose everything you have done since the last time you saved. If you really want to quit **vi** without saving your work, type:

```
:q!
```

Searching for Strings

In a large document, searching for a particular text string can be very time consuming. The **/** command prompts for a string to search for in the file. When you press <Enter>, **vi** searches the file for the next occurrence of the string you entered.

To try searching for a string, first move to the top of **vitest**. Then type:

```
/Blu
```

and press <Enter>.

As soon as you enter **/**, it is displayed on the bottom of the screen. As you type the string **Blu**, it is echoed at the bottom of the screen. You can use <Backspace> to fix mistakes as you type the search string. After you press <Enter>, the cursor moves to the first occurrence of the string.

The **n** command searches for the next occurrence of the last string you searched for. Try it now by entering:

```
n
```

The cursor should move to the next occurrence of the string, which is the **th** in the word **with**. You can also use **N** like **n** to search the other direction through the file.

If you just type a slash without anything after it, **vi** looks for the most recent word or phrase you searched for.

Searching Backwards through a File

To specify a search string for a backward search through the file, use the **?** command in the same way as **/**. If you just type a **?** without anything after it, **vi** searches backwards for the most recent word or phrase you searched for. When you search backward, the **n** command moves the cursor backward to the next occurrence of the string, and the **N** command moves the cursor forward.

Case-sensitive Searching

When you type in characters after a slash or question mark, make sure you enter them in the correct case. For example, ask **vi** to search for `IN`, and type the word in uppercase. You will see that **vi** prints the message `Pattern not found` at the bottom of the screen. As it turns out, this file does not contain the word `IN` in uppercase, although it has the word several times in lowercase.

Notice that the message used the word `Pattern`. In a **vi** command, anything after a slash or question mark is called a pattern.

Special Search Characters

In order to make searching more useful, **vi** gives special meanings to several characters when they are used in patterns. For example, the circumflex or caret character (^) stands for the beginning of a line. Move the cursor to the next line and type:

```
/^All
```

vi will look for the word `All` occurring at the beginning of a line.

The end of a line is represented by the dollar sign ($). Move the cursor to the next line and type:

```
/plum$
```

You will see that **vi** searches forward for a line that ends in the word `plum`.

Inside patterns, the dot (.) stands for any character. For example, move the cursor to the top of the file and type:

```
/t.e
```

You will see that the cursor moves to the word `the`. Type / over and over, and you will see the cursor keep jumping forward to any sequence of three letters that starts with `t` and ends in `e`. Were you surprised that the cursor jumped into the middle of the word `slithey`? **vi** finds character strings, even when they are in the middle of larger words.

Inside patterns, a dot followed by an asterisk (`.*`) stands for any sequence of zero or more characters. For example, type:

```
/^A.*g$
```

You will find the next line that begins with the letter `A`, ends with the letter `g` and has any number of characters in between.

Character	Stands for:
`^`	Beginning of the line
$	End of the line
.	Any character
.*	Any sequence of zero or more characters

vi gives special meanings to several other characters inside patterns. For complete details, see the appendix on "Regular Expressions" in *OS/390 UNIX System Services Command Reference*. A regular expression is the POSIX name for a pattern; here we use the word `pattern` because it is more descriptive.

What happens if you want to search for a character that has a special meaning in patterns? For example, suppose you want to search for the string `2.3*25` somewhere in a file. If you just type:

```
/2.3*25
```

vi will think the `3*` stands for zero or more occurrences of the digit `3`, not the `*` character. In such cases, put a backslash (\) in front of any characters with special meanings, as in the example:

```
/2\.3\*25
```

Notice that we had to put a backslash in front of the dot as well as the asterisk; both have a special meaning in patterns.

By default, all searches in **vi** wrap around from the bottom of the file to the top. Similarly, if you use question marks to search backward through a file, the search will wrap around from the top of the file to the bottom, if necessary.

Moving Text

The first step to moving a block of text is to select text for moving. In fact, you already know how to do this. The **d** command not only deletes a block of text but also copies it to a paste buffer. Once in the paste buffer, the text can be moved by repositioning the cursor and then using the **p** command to place the text after the current cursor position.

To delete the first line of the file, move there and type:

```
dd
```

The line is deleted and copied into the paste buffer, and the cursor is moved to the next line in the file. To paste the line following the current line, type:

```
p
```

To paste text before the cursor rather than after it, use the **P** (uppercase) command.

If you delete a letter or word size block, it will be pasted into the new position within the current line. For example, to move the word **came** after the word **spider**, you could use the following command sequence:

```
/came <Enter>
dw
/spider <Enter>
p
```

Copying Text

You copy text in the same manner as you move it, except that instead of using the delete text command **d**, you use the yank text command, **y**. The **y** command copies the specified text into the paste buffer without deleting it from the text. It follows the same syntax as the **d** command. You can also use the shortcut **yy** to copy an entire text line into the paste buffer, in the same way as **dd**.

For example, you can copy the first two lines of the file to a position immediately underneath them. To do so, enter the following command sequence from the first line of the file:

```
2yy
j
p
```

Note that you must move down one line using **j** or the two lines will be pasted after the first line rather than after the second.

Other vi Features

Here are a few more helpful **vi** subcommands:

J Joins the following line to the current line

. Repeats the last command

s Substitutes the current character with the following entered text

x Deletes the current character

Message: ″vi/ex edited file recovered″

Have you received mail with this subject: ″vi/ex edited file recovered″ ? This is what the mail messages look like:

```
From OMVS Mon Apr 29 13:58:50 1996
To: 1234567
Status: R
Subject: vi/ex edited file recovered.

Mon Apr 22 13:47:45 1996, the file

        "NoFilename"

that you were editing has been recovered.
You can retrieve most of your changes to this file
using the "-r" option or the ":recover" command of the
vi or ex editors. An easy way to do this is with the command

          vi -r NoFilename
```

This is happening because you used the **vi** command when you logged into the shell with the OMVS command. OMVS is a line-mode terminal interface which does not support curses (raw mode), and **vi** uses curses. To use **vi**, you need to login using **rlogin**, **telnet**, or OS/390 UNIX System Services Communications Server.

When **vi** is invoked, it first creates files in **/tmp** so that it can recover the file being edited if any system errors occur. When **vi** is invoked from OMVS, it creates its recovery files in **/tmp** but cannot continue.

The current default directory for temporary **vi** files (usually **/tmp**) may be implemented as a TFS. In this case, all **vi**'s temporary files that the **exrecover** daemon uses for recovery would be gone after a system crash. In Release 8, the

environment variable **TMP_VI** can contain a directory pathname that can be specified by an administrator as an alternative location for these temporary files. See “Using the TMP_VI Environment Variable” for more information.

The **exrecover** command automatically recovers these files. By default this command is started from the **/etc/rc** file. In **/etc/rc** you will see these lines:

```
# Invoke vi recovery
mkdir -m 777 /etc/recover
/usr/lib/exrecover
```

Every IPL, the **/etc/rc** script is run and the **exrecover** command is also run. **exrecover** goes through all the recovery files that **vi** left in **/tmp**. They are named ″VIsomething″, and there are 3 of them created for each **vi** command. **exrecover** creates directories in **/etc/recover** for each userid, puts the recovered files there, and sends the user mail telling what it did.

Using the TMP_VI Environment Variable

IBM recommends that this environment variable be set by a system administrator rather than a user. If a user sets the **TMP_VI** directory to something other than the name that **exrecover** recognizes as **TMP_VI**, the user must run the **exrecover** daemon manually to allow the directory files to be converted to the recoverable files used by **vi** (located in /**etc/recover/$LOGNAME**).

Note: A system administrator should NOT set **TMP_VI** to **/etc/recover/$LOGNAME** or set **TMP_VI** to any directory where a pathname component is an environment variable with a user’s value different than the value of the init process, for example, $HOME.

The temporary **vi** files are converted into a form that is recoverable by **vi** when **exrecover** is run during IPL. Because **exrecover** is issued during IPL, it is owned by the init process and therefore contains different values for certain environment variables if those environment variables have been set. Throughout the file system, there may be some temporary files that can be converted only by **exrecover**. This conversion can be done manually by a system administrator to recover files owned by all users or by a single user to recover their own files.

Stopping the Mail Messages

If no one at your installation intends to use **vi**, a superuser can get rid of the exrecover mail messages as follows:

1. Edit **/etc/rc**
2. Comment out the line that says ″/usr/lib/exrecover″ This stops the exrecover command from running, so no new mail messages will be sent.
3. `cd /tmp`
4. `rm VI*`

If your installation has some users who will be editing with **vi**, then it’s a little trickier. In this case, your **vi** users will want the recovery capabilities of **vi**, so you do not want to remove the exrecover command from **/etc/rc**.

Anyone can remove those **/tmp/VI*** files that were generated when users on dumb terminals tried **vi**. To stop **exrecover** from sending new mail messages about those files:

1. Broadcast a message to make sure no one is using **vi** at the moment
2. `cd /tmp`

3. `rm VI*`

Deleting the Old mail Messages

If you want to only delete only the mail messages sent by **exrecover**:

1. Enter

   ```
   mailx
   ```

2. Use the **mailx** commands to read each message: Enter the number of the message
3. Enter

   ```
   d
   ```

   to delete that message.

To delete all your mail messages, do this:

1. `rm /usr/mail/$LOGNAME`

   But be careful because this will delete all your mail messages!

Using the ed Editor

Using the Shell:

ed is a line editing program available in the shell for editing text files. When you use **ed** to edit a file, the file is copied into the *edit buffer,* a temporary storage area. You use various subcommands to edit the text in the buffer. When you end your edit session, the contents of the buffer are written to the file system, overwriting the previous contents of the file.

With **ed**, you work with one line in the buffer at a time. In this discussion, that position in the buffer is called the *current working line*.

For more details about **ed**, see *OS/390 UNIX System Services Command Reference*.

Creating and Saving a Text File

1. To begin editing a new file, enter:

   ```
   ed filename
   ```

   where *filename* is the name of a new file.
2. After you see the ?*filename* message, enter:

   ```
   a
   ```

   This indicates you want to *append* lines.
3. Type your text. At the end of each line, press <Enter>. You can then enter more text.
4. When you are finished entering text, enter:

   ```
   .
   ```

   (a period) at the start of a new line.
5. To write the contents of the edit buffer to the file *filename*, enter:

   ```
   w
   ```

After writing to the file, the shell displays the number of characters copied—for example, 746. This number includes blanks and newline characters appended to each line of text, which you cannot see on the screen.

If you want to write to a file different from the original *filename*, specify a different filename when you enter the **w** subcommand; for example:

```
w diffname
```

Entering the **w** subcommand does not change the contents of the buffer.

6. To exit the **ed** program, enter:

   ```
   q
   ```

   This deletes the contents of the buffer.

Editing an Existing File

To begin editing an existing file, enter:

```
ed filename
```

Your current working line is the last line in the file. If you want to change your position in the file before you begin editing, see “Identifying Line Numbers and Changing Your Position in the Buffer”.

If you are already using **ed**, have finished editing one file and saved it with the **w** subcommand, and you now want to edit another file, enter:

```
e filename
```

This erases the previous contents of the buffer and loads in the new file.

Identifying Line Numbers and Changing Your Position in the Buffer

To find out how many lines there are in a file, enter:

```
$=
```

To identify the line number of your current working line, enter:

```
.=
```

You can make a different line in the file your current working line and then identify its number.

To move the current working line forward a line at a time, press <Enter>. The text of the line is displayed.

To move the current working line backward a line at a time, enter:

```
–
```

(hyphen). The text of the line is displayed.

Changing Position Using Numbers

To change the current working line to a different line in the file, enter:

```
n
```

where *n* is the number of the line you want to work with. The text of the line is displayed.

To move the current working line *n* lines forward, enter:

```
.+n
```

To move the current working line *n* lines backward, enter:

```
.–n
```

Changing Position Using a Search String (Regular Expression)

If you don't know the number or position of the line you want to make your current working line, you can locate a string (or *regular expression*) in the line. To search forward for one or more words or a string of characters, enter:

```
/regexp/
```

where *regexp* is one or more words or a string of characters. The line containing the search string is displayed and it is now your current working line.

To search backward for one or more words or a string of characters, enter:

```
?regexp?
```

where *regexp* is one or more words or a string of characters. The line containing the search string is displayed and it is now your current working line.

Appending One File to Another

If you want to append a file at the end of the file you are working on in the buffer, enter:

```
r filename
```

Or, if you want to read a file in after a specific line in the buffer, enter:

```
nr filename
```

where *n* is the number of the line in the file.

To display the contents of a file in the edit buffer, enter:

```
,p,
```

On your screen, each line of the file is displayed, for example:

```
,p,
Oh, you better watch out
You better not shout
You better not cry
I'm telling you why
```

Once you know the line numbers, you could insert the file **scrooge** after the line `You better not cry`. Thus, you would enter:

```
3r scrooge
```

Displaying the Current Line in the Edit Buffer

When you enter subcommands you identify the current working line with the symbol `.` (dot).

To display the current working line, enter:

```
p
```

To display the line number of the current working line, enter:

```
.=
```

Changing a Character String

For changing text or correcting spelling errors, use the **s** (substitute) subcommand. When you enter the subcommand, the line you are changing becomes your current working line. To display the line after you make the change, enter the **p** (print) subcommand.

- To substitute text for the first matching string on the current working line, enter:
  s/*oldtext*/*newtext*/
- To substitute text for the first matching string on a specified line, enter:
  *n*s/*oldtext*/*newtext*/

  where *n* is the number of the line.
- To substitute text for the first matching string on more than one line, enter:
  a1,*a2*s/*oldtext*/*newtext*/

  where *a1* is the number (or "address") of the first line to be changed and *a2* is the number of the last line to be changed.
- To change every occurrence of a string on more than one line, enter:
  a1,*a2*s/*oldtext*/*newtext*/g

  where *a1* is the number of the first line to be changed and *a2* is the number of the last line to be changed. **g** is the global operator.

  To change every occurrence of a string on one line, enter:
  *n*s/*oldtext*/*newtext*/g

  g is the global operator.
- To delete a word or string, enter:
  s/*oldtext*//

Inserting Text at the Beginning or End of a Line

Using the **s** (substitute) subcommand and these two special substitution characters, you can insert text at the beginning or end of a line:

^ (circumflex)
: Inserts text at the beginning of the line.

$ (dollar sign)
: Inserts text at the end of the line.

- To insert text at the beginning of the current working line, enter:
  s/^/*newtext*
- To insert text at the beginning of a specified line, enter:
  *n*s/^/*newtext*

  where *n* is the number of the line. This line becomes the current working line.
- To insert text at the end of the current working line, enter:
  s/$/*newtext*
- To insert text at the end of a specified line, enter:
  *n*s/$/*newtext*

  where *n* is the number of the line. This line becomes the current working line.

Deleting Lines of Text

Use the **d** (delete) subcommand to delete one or more lines of text. After you delete a line, the first line following the deleted line (or lines) becomes the current working line. After a line is deleted, the remaining lines in the buffer are renumbered.

- To delete the current working line, enter:

  ```
  d
  ```

- To delete a specific line number, enter:

  ```
  nd
  ```

  where *n* is the line number.

- To delete more than one line, enter:

  ```
  a1,a2d
  ```

  where *a1* is the number of the first line and *a2* is the number of the last line.

Changing Lines of Text

To replace one or more lines with one or more new lines, use the **c** (change) subcommand. This actually deletes the lines you want to replace and then inserts the new lines.

1. Enter:

   ```
   a1,a2c
   ```

   where:
   a1 is the number of the first line to be deleted.
   a2 is the number of the last line to be deleted.
2. Type the new lines, pressing <Enter> at the end of each line.
3. End the insert by typing a **.** (period) on a line by itself.

Inserting Lines of Text

To insert one or more lines of new text into the edit buffer, use the **i** subcommand.

1. You can specify the subcommand in one of two ways, depending on how you want to identify the line that the new lines are to be inserted *before*:
   - If you know the number of the line that you want to insert the new lines before, enter:

     ```
     ni
     ```

     where *n* is the number of that line.
   - To identify the line that the new lines are to be inserted before by words or a string of characters in the line (known as a regular expression), enter:

     ```
     /regexp/i
     ```

   where *regexp* is one or more words or a string of characters.
2. Enter the new lines.
3. End the insert by typing a **.** (period) on a line by itself.

Copying Lines of Text

You can copy one or more lines within the edit buffer, using the **t** (transfer) subcommand.

To copy one line, enter:

```
a1tn
```

where:
a1 is the number of the line to be copied.
n is the number of the line that the line is to be copied after.

To copy a block of lines, enter:

```
a1,a2tn
```

where:
a1 is the number of the first line in the block of lines to be copied.
a2 is the number of the last line in the block of lines to be copied.
n is the number of the line that the lines are to be copied after.

To copy lines to the top of the edit buffer, use `0` as the line number for the lines to be copied after.

To copy lines to the bottom of the edit buffer, use `$` as the line number for the lines to be copied after.

Moving Lines of Text

Use the **m** (move) subcommand to move a block of lines to a different position in the edit buffer. After the text is moved, the last line in the block of lines becomes the current working line. Enter:

```
a1,a2mn
```

where *a1* is the number of the first line in the block, *a2* is the number of the last line in the block, and *n* is the number of the line that the block of lines are to be moved *after*.

To move text to the top of the buffer, use `0` as the line number for the lines to be moved after.

To move text to the end of the buffer, use `$` as the line number for the lines to be moved after.

Undoing a Change

To “undo” a change, use the **u** subcommand. This subcommand undoes the changes made by the last subcommand that changed the buffer. For the purposes of **u**, subcommands that change the buffer are: **a**, **c**, **d**, **g**, **G**, **i**, **j**, **m**, **r**, **s**, **t**, **v**, **V**, and **n**.

Entering a Shell Command while Using ed

To temporarily switch out of the **ed** program and run a shell command, enter:

```
!command name
```

Ending an ed Edit Session

When you have finished working with a file, you save the changes by entering:

```
w
```

To end the edit session, enter:

```
q
```

If you enter **q** without entering **w** to save the buffer first, the changes you have made are not saved.

Default Permissions

When you create a file using the **ed** editor, its default permissions are
owner=rw-
group=rw-
other=rw-

The octal number is 666.

Using sed to Edit an HFS File

Using the Shell:

sed is a *noninteractive* editor. This means that you do not use it in an interactive session. You enter the **sed** command specifying a file containing editing commands and a data file and it produces an edited target file with no user interaction. **sed** is intended for *systematic* editing, as opposed to the usual *editing-on-the-fly* performed by interactive users.

sed subcommands are similar to those used with **ed**, except that **sed** commands view the input text as a stream rather than as a directly addressable file. Each line of the file containing editing commands has up to two addresses, a single-letter command, possible command modifiers, and an ending newline character.

For more details on **sed**, see *OS/390 UNIX System Services Command Reference*.

Chapter 19. Printing Files

If you are a workstation user, you are probably accustomed to having a printer close by, if not on your desk. In contrast, the MVS system intentionally screens the user from printer knowledge and uses a printer resource pool. One facility provided to manage this pool is the System Display and Search Facility (SDSF).

You can, of course, download HFS files and print them at your workstation. However, it may be more convenient to have print jobs sent to accessible Job Entry Subsystem (JES) printers directly by the shell. In addition, you may want to use the large-volume printing facilities offered by MVS.

Additional information on printing in the UNIX environment can be found in the Printing from OS/390 UNIX System Services section of the *OS/390 Infoprint Server User's Guide*

Formatting Files for Online Browsing or Printing

Using the Shell:

Using shell commands, you can format a file in a certain way for browsing or printing. Later, with the **lp** command, you can send the formatted file to be printed.

If you want to format and print a file immediately, you can request this printing as a single piped command.

To format an HFS file, use the **pr** command; for example:

```
pr -2 report1
```

This command requests the shell to format for printing in two columns a file named **report1**, sending the output to standard output (your workstation screen). The file appears in the format you selected on your screen. There are many format options for the **pr** command, as described in *OS/390 UNIX System Services Command Reference*.

If, instead, you had redirected standard output to a file named **report2**, you could later print the file by entering:

```
lp report2
```

This would request the printing of the formatted file in **report2**; because the *dest* option is not specified, the file is sent to the default printer destination.

If you want to format a file and print it right away, you can join the requests using a pipe. (See "Using a Pipe" on page 75 for more information on using a pipe.) For example:

```
pr -2 report1 | lp
```

formats and prints the file **report1**.

To save the formatted output as well as print it, try:

```
pr  -2 report1 | tee report2 | lp
```

This command formats **report1**, pipes the formatted output to **tee**, which writes it to **report2** and at the same time pipes **report2** to the next command, **lp**, which sends the input to the printer queue. The formatted output is saved in **report2**.

Printing Requests in Shell Scripts

Including print requests in a shell script may limit the portability of the shell script because printer configuration options in other operating systems may differ. To minimize the work to port the shell script to another system, be sure to identify environment assumptions and aliases that may have been used.

Printing with the lp Command

Using the Shell:

You can use the **lp** command to send a previously formatted file to a JES printer:

```
lp filename
```

You can specify more than one filename with the command. The **lp** command uses existing JES printer facilities. Because a default printer destination is assigned to you, you do not need to specify a destination (with the **-d** *dest* option) when entering the **lp** command. However, you can specify a destination other than the default by using the **-d** *dest* option. For **-d** *dest*, you can specify LOCAL for any printer or any of the symbolic destination names your system programmer has defined for JES printers. These symbolic names are defined locally.

Class is a frequently used option, and at your site there may be several different classes defined. For instance, `C` may be designated the class for confidential information. Suppose you want to print the file **temp.prt** using the default printer destination and specifying class `C`; you would enter it in either of these ways:

```
lp -d ,c temp.prt
```

```
lp -d,c temp.prt
```

The parameters on the **-d** option are positional, so if you omit a destination, you must still include the comma.

To specify the number of copies you want printed, use the **-n** option. For example,

```
lp -n 2 report2
```

requests the printing of two copies of the formatted file in **report2** to the default printer destination.

If you have OS/390 Print Server installed on your system, then you will use the Print Server version of the **lp** command. For more information about OS/390 Print Server, see *OS/390 Infoprint Server User's Guide*.

Printing with TSO/E Commands

Using TSO/E:

There are some printer services not available through the **lp** command, such as printing a single file to multiple destinations. To print in TSO/E, you need to know:

- The TSO/E commands you can use to submit print jobs
- The printing options (class) you want to specify

Here are the steps:

1. If you are working in the shell, switch to TSO/E command mode by pressing the TSO function key.
2. If you want to print an MVS data set, skip to the next step. If you want to print an HFS file, you must first copy it into an MVS data set using the TSO/E OGET or OCOPY command. (See “Copying an HFS File into a Sequential Data Set or PDS Member” on page 286 for more information on copying.)

   Note: Someone at your installation may, in fact, have written an MVS command list (CLIST) or a REXX program that you can enter as a TSO/E command for printing. The command list could include the OGET or OCOPY command and would let you specify such things as multiple destinations, special character sets, and notification for a set of people.
3. You can format an MVS data set for printing using TSO/E commands. Possibly you will be using ISPF panels.
4. Print the data set:
   - To enter the request to print the formatted data set, for example, you might enter:

     ```
     printds da(project1.list) class(c)
     ```
   - To submit a print request to the MVS job queue, for example, you might enter:

     ```
     submit  jcl.cntl(print1)
     ```

For a print batch job request, the system returns a message confirming that the job request has been received.

Checking the Status of Print Jobs

If you submit a print job with a shell command, there is no way to check on the status of the job. (The **lpstat** and **cancel** commands are not supported.) All output looks the same on the queue in terms of job number. Print jobs could have different setups such as destination or class, but normally about the only difference is number of lines, bytes, or pages and, of course, the time of day the output was available to print.

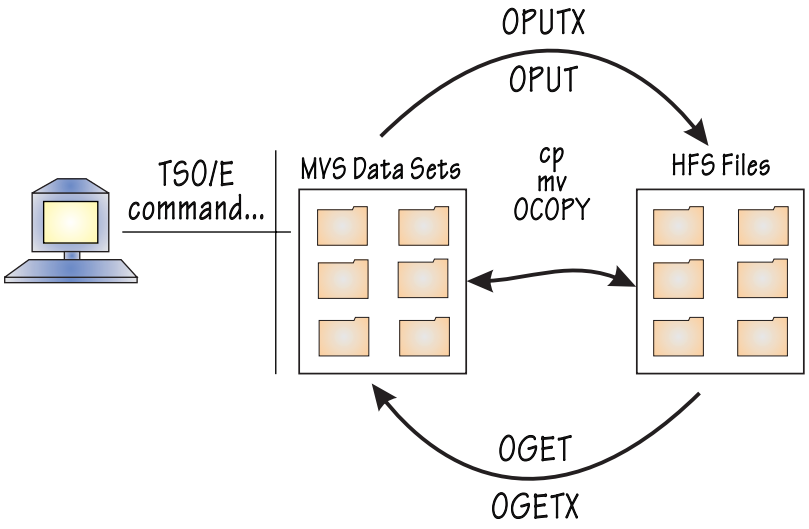
Note: The Print Server is included with OS/390. The Print Server, if enabled, replaces the **lp** command and provides other commands, including **lpstat** and **cancel**.

If your operating system includes SDSF, you can use the SDSF panels to monitor and control a TSO/E or batch print job, look at its output as it is running, check its completion, and release it to print.

For a batch job, the STATUS command can provide status if you specify the job name as your user ID followed by one character (for example, MACNEILA). You cannot use the STATUS command for print jobs that you ran using **lp** or PRINTDS. STATUS takes either no operands or one or more job names as operands. If you use no operands, the system looks for jobs with names that start with your user ID followed by one character. If you list a job name, it looks for that job name.

Chapter 20. Copying Files Between UNIX Files and MVS Datasets

You can copy files using UNIX System shell **cp** and **mv** commands or with TSO/E **OPUT, OPUTX, OGET, OGETX,** and **OCOPY** commands.



Copying Data Using UNIX System Shell Commands

You can use the UNIX System shell **cp** and **mv** commands to copy or move:

- MVS partitioned data set members (PDS or PDSE) to a file system.
- UNIX files to an MVS partitioned data set member (PDS or PDSE).
- MVS sequential data sets to the file system.
- UNIX files to an MVS sequential data set.
- MVS partitioned data sets into a file system directory.
- An MVS sequential data set to another MVS sequential data set.
- A partitioned data member into another partitioned data set member.

Note: With the **cp** and **mv** shell commands you can specify whether the file or data set is to be copied/moved as text, binary, or executable. You can also append or truncate suffixes.

For more information about the **cp** and **mv** shell utilities, refer to *OS/390 UNIX System Services Command Reference*.

Examples

The following examples use the **cp** command. The same syntax applies with **mv**.

1. To specify **–P params** for a non-existing sequential target:

   ```
   cp -P "RECFM=U,space=(500,100)"file "//'turbo.gammalib'"
   ```

2. To copy file **f1** to a fully qualified sequential data set `'turbo.gammalib'` and treat it as a binary:

   ```
   cp -F bin f1 "//'turbo.gammalib'"
   ```

3. To copy all members from a fully qualified PDS `'turbo.gammalib'` to an existing UNIX directory `dir`:

```
cp "//'turbo.gammalib'" dir
```

4. To drop .c suffixes before copying all files in UNIX directory `dir` to an existing PDS `'turbo.gammalib'`:

```
cp -S d=.c dir/* "//'turbo.gammalib'"
```

Copying Data Using TSO/E Commands

You can also use TSO/E commands to copy data between HFS files and MVS data sets:

- MVS sequential data sets, entire partitioned data sets or members, entire PDSEs or members, into the file system.
- HFS files or directories into MVS data sets (sequential data set, partitioned data set, or PDSE).
- HFS files or directories into workstation files or directories.
- Workstation files or directories into HFS files or directories.

The TSO/E commands for doing this are:

OPUT Puts (copies) an MVS sequential data set or partitioned data set member into the file system. You can specify text or binary data, and select code page conversion for singlebyte data.

OPUTX
Puts (copies) a sequential data set, a data set member, an MVS partitioned data set, or a PDSE into an HFS directory. You can specify text or binary data, select code page conversion for singlebyte data, specify a copy to lowercase filenames, and append a suffix to the member names when they become filenames.

OGET Gets an HFS file and copies it into an MVS sequential data set or partitioned data set member. You can specify text or binary data, and select code page conversion for singlebyte data.

OGETX
Gets an HFS file or directory and copies it into an MVS partitioned data set, PDSE, or sequential data set. You can specify text or binary data, select code page conversion for singlebyte data, allow a copy from lowercase filenames, and delete one or all suffixes from the filenames when they become PDS member names.

OCOPY
Copies data in either direction between an MVS data set and an HFS file, using ddnames. OCOPY can also copy within MVS (one data set to another data set) or within the shell (one file to another file). OCOPY has a CONVERT operand for converting singlebyte data from one code page to another.

There is also an MVS utility for copying, BPXCOPY, which has similar function as OPUT, with some differences.

To read about:

- The TSO/E OPUT, OPUTX, OGET, OGETX, and OCOPY commands, and the BPXCOPY utility, see *OS/390 UNIX System Services Command Reference*.
- The TSO/E ALLOCATE and FREE commands, see *OS/390 TSO/E Command Reference*. Both of these commands have OS/390 UNIX keyword parameters. There are several examples of the ALLOCATE command in this chapter. After

you have finished copying to or from a data set, it is a good idea to free the allocated data set with the FREE command.

Executable Modules: You may also want to copy executable modules between MVS data sets and the file system. For information on how to do that, see “Copying Executable Modules between MVS and the File System” on page 293.

Copying Data: Code Page Conversion

The method used to convert data from one code page to another depends on whether it is singlebyte or doublebyte data.

Singlebyte Data

If you are copying singlebyte data into or out of the HFS, you can use one of these:

- Working in MVS, you can use the OS/390 c/c++ **iconv** utility to convert MVS data from one code page to another. For information on the OS/390 c/c++ **iconv** utility, see *OS/390 C/C++ Programming Guide*.
- Working in the shell, you can use the **iconv** shell command to convert HFS data from one code page to another. For information on **iconv**, see *OS/390 UNIX System Services Command Reference*.
- The CONVERT operand on each of the commands OCOPY, OGET, OGETX, OPUT, and OPUTX provides these code page conversion choices for the data as you are copying:

  CONVERT((BPXFX111))
  Specifies a conversion table to convert between code pages IBM-037 and IBM-1047.

  CONVERT((BPXFX311))
  Specifies an ASCII-EBCDIC conversion table to convert between code pages ISO8859-1 and IBM-1047.

  CONVERT(YES)
  Specifies the default conversion table BPXFX000, which is an alias that points to BPXFX111, to convert the data.

  CONVERT(user-defined table)
  You can specify the name of a user-defined conversion table with the CONVERT operand.

  In the above list, the use of (( )) and no data set name indicates that you are specifying a member that is a module in the standard search order for MVS.

Doublebyte Data

If you are moving doublebyte data into or out of the HFS, you can convert the data to or from the shell-supported DBCS code page IBM-939 using one of two utilities:

- Working in MVS, you can use the OS/390 c/c++ **iconv** utility. For information on the OS/390 c/c++ **iconv** utility, see *OS/390 C/C++ Programming Guide*.
- Working in the shell, you can use the **iconv** shell utility. For information on the **iconv** shell utility, see *OS/390 UNIX System Services Command Reference*.

Example: Using the iconv Shell Utility with MBCS Data

In this example, the PDSE member MBCSDATA is moved into the file system and then converted to code page IBM-939 from code page IBM-932 (a multibyte ASCII code page):

1. Run the OPUT command from the shell, using the double quotation marks to prevent the shell from processing it:

   ```
   tso oput "'usr3.data(mbcsdata)' '/tmp/usr3/mbcsdata' bin"
   ```

2. Change to the directory that the file **mbcsdata** is in:

   ```
   cd /tmp/usr3
   ```

3. Use **iconv** to convert the data and put it into the output file **dbcsdata**:

   ```
   iconv -f IBM-932 -t IBM-939 mbcsdata > dbcsdata
   ```

Copying a Sequential Data Set or PDS Member into an HFS File

You might want to copy an MVS sequential data set or a member of a partitioned data set or PDSE to an HFS file, so that:

- The data can be used by a program running under the shell.
- If it is a C program source file developed at your workstation, you can compile, link-edit, and debug it in the shell using the **c89/cc/c++** and **dbx** commands.

The data set can be text or binary. If you are moving the data set permanently to the file system, use the TSO/E DELETE command to delete the data set after copying it.

Use the TSO/E OPUT command or OCOPY commands to do the copy. You can enter either command:

- In TSO/E, in the shell, or in ISPF. See “Entering a TSO/E Command” on page 202 for information on entering TSO/E commands in TSO/E, the shell, and ISPF.
- In batch, using a Terminal Monitor Program (TMP) job.

To specify data set names and filenames, use the OPUT command. To specify ddnames, use the ALLOCATE command and the OCOPY command together. Because you can specify permissions on the ALLOCATE command first, the OCOPY command lets you set the permission bits for a newly created file.

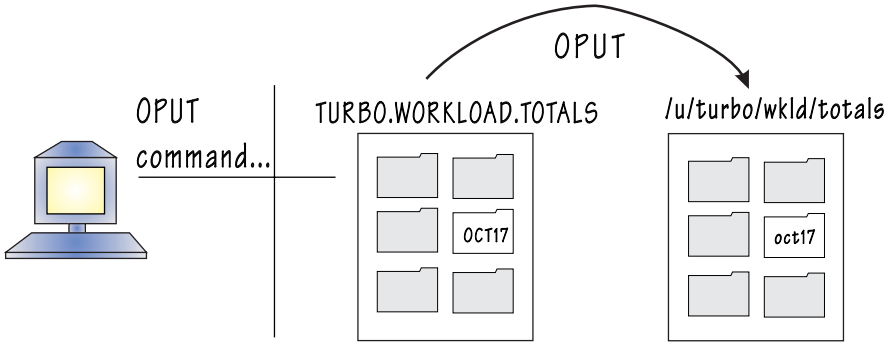
OPUT

The OPUT command syntax is:

```
OPUT mvs_data_set_name[(member_name)]
     'hfs_file_name'
     {TEXT | BINARY}
     {CONVERT(convert_table_name | YES | NO)}
```

You can use the CONVERT option for singlebyte data, but not for doublebyte data. See “Doublebyte Data” on page 279 for information on code page conversion for doublebyte data.

Example: Using OPUT with a PDSE Member



If the user ID TURBO wants to copy a member of a PDSE into a file, TURBO enters the following TSO/E OPUT command:

```
OPUT WORKLOAD.TOTALS(OCT17) '/u/turbo/wkld/totals/oct17' TEXT CONVERT(YES)
```

This command:

- Copies the MVS partitioned data set member OCT17 from the data set TURBO.WORKLOAD.TOTALS to a text file with the pathname **/u/turbo/wkld/totals/oct17**.
- Converts the data using the default conversion table (from MVS code page IBM-037 to code page IBM-1047), because YES was specified. To use a different conversion table, specify its name—for example, BPXFX311 for conversion to and from the ASCII conversion table. If you do not want conversion, omit the CONVERT operand or specify CONVERT(NO).

  For more information, see “Copying Data: Code Page Conversion” on page 279.
- Sets a default mode (read-write-execute permission) if **oct17** is a *new* file. For a new text (non-U-format data set) file, the default is octal 600:

  owner=rw-
  group=---
  other=---

  The default mode for a binary load module (U-format data set) is octal 700:

  owner=rwx
  group=---
  other=---

  After the file is created, you can change the permissions with the **chmod** command.

If there is an existing HFS file with the pathname that you specify on the command, it is replaced automatically and the mode of the file is not changed.

The directories specified in the pathname must already exist. This command creates a new file, but it does not create a new HFS directory.

Example: Using OPUT with a Sequential Data Set

If the user ID TURBO wants to copy a sequential data set into a file, TURBO enters the following TSO/E OPUT command:

```
OPUT WORKLOAD.PROJA.NOV '/u/turbo/wkld/proja/nov' TEXT CONVERT(YES)
```

This command:

- Copies the MVS sequential data set TURBO.WORKLOAD.PROJA.NOV to a text file with the pathname **/u/turbo/wkld/proja/nov**.
- Converts the data from the MVS code page IBM-037 to code page IBM-1047, using the default conversion table because YES was specified.
- Because **proja** is a *new* text file, sets a default mode (read-write-execute permission) of octal 600, representing:
  owner=rw-
  group=---
  other=---

OCOPY

To copy a data set into a file and use data definition names (ddnames) instead of a data set name and pathname, use the OCOPY command:

```
OCOPY INDD(ddname1) OUTDD(ddname2)
      {TEXT | BINARY}
      {CONVERT(convert_table_name | YES | NO)}
      {TO1047 | FROM1047}
      {PATHOPTS(USE | OVERRIDE)}
```

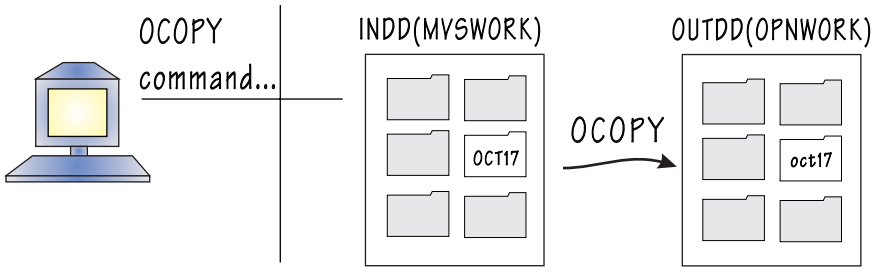
1. If the data set and file are not yet allocated, allocate them and specify ddnames, either using the ALLOCATE command or the DD statement in JCL.

   The ALLOCATE command has these operands for allocating an HFS file:

   ```
   PATH
   PATHDISP
   PATHMODE
   PATHOPTS
   ```

   They are explained in *OS/390 TSO/E Command Reference*.
2. Enter the OCOPY command, being sure the ddnames used match the ddnames specified when the data set and file were allocated.
3. You can use the CONVERT option for singlebyte data, but not for doublebyte data. See “Doublebyte Data” on page 279 for information on code page conversion for doublebyte data.
4. If you are moving the data set or partitioned data set member permanently to the file system: After the copy is completed, delete the original using the TSO/E DELETE command.

Example: Using ALLOCATE and OCOPY



1. Using the ALLOCATE command to associate the PDSE member with the ddname specified in the DDNAME keyword, user TURBO could enter:

```
ALLOCATE DDNAME(MVSWORK) DSNAME('TURBO.WORKLOAD.TOTALS(OCT17)')
```

Note: For an ALLOCATE that begins with your TSO/E prefix as the high-level qualifier, you can enter the data set name more simply as DSNAME(WORKLOAD.TOTALS(OCT17))—without the user ID shown above. (The TSO/E prefix defaults to your user ID, but can be set with the PREFIX command.) If you do not enclose the data set name in quotes. TSO/E automatically prefixes the name with your TSO/E prefix. For JCL, you need the user ID.

2. Using the ALLOCATE command to create a new HFS file and associate it with the ddname specified in the DDNAME keyword, TURBO could enter:

```
ALLOCATE DDNAME(OPNWORK) PATH('/u/turbo/wkld/totals/oct17')
         PATHDISP(KEEP,DELETE) PATHOPTS(ORDWR,OCREAT)
         PATHMODE(SIRUSR,SIWUSR)
```

   In this example:

   - PATHDISP(KEEP,DELETE) indicates that the file should be saved if the session ends normally, but it should be deleted if the session ends abnormally.
   - The PATHOPTS operand is required only when you are creating a new file. PATHOPTS(ORDWR,OCREAT) indicates the owner has read/write access and this is a new file being created.
   - Specifying PATHMODE is required only when you are creating a new file (OCREAT). PATHMODE(SIRUSR,SIWUSR) indicates the owner has read and write permission. If you do not specify a PATHMODE, the default permissions set when the file is allocated are:
     owner=---
     group=---
     other=---

3. After the data set and file have been allocated, TURBO would enter the OCOPY command, using the ddnames, to copy the MVS partitioned data set member to an HFS file using the default conversion table:

```
OCOPY INDD(MVSWORK) OUTDD(OPNWORK) TEXT CONVERT(YES) PATHOPTS(USE)
```

   PATHOPTS(USE) indicates TURBO wants to use the PATHOPTS specified on the ALLOCATE command.

Example: Using JCL and OCOPY

Alternatively, TURBO could specify the ddnames in the DD statements and perform the OCOPY in the JCL for a batch job. A DD statement allocates a data set or file and sets up a ddname. In the following example, the //INMVS statement refers to the input data set, and the //OUTHFS statement refers to the output file. For example:

```
//TEST JOB MSGLEVEL=(1,1)
//COPYSTEP EXEC PGM=IKJEFT01
//INMVS    DD DSN=TURBO.WORKLOAD.TOTALS(OCT17),DISP=SHR
//OUTHFS   DD PATH='/u/turbo/wkld/totals/oct17',
//            PATHDISP=(KEEP,DELETE),
//            PATHOPTS=(OWRONLY,OCREAT,OEXCL),PATHMODE=(SIRUSR,SIWUSR)
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
OCOPY INDD(INMVS) OUTDD(OUTHFS) TEXT CONVERT(YES) PATHOPTS(USE)
/*
```

In this example:

- IKJEFT01 is the name of the Terminal Monitor Program (TMP), which needs to be started to process the TSO/E OCOPY command.
- For CONVERT(YES), the default is TO1047 when you are copying from an MVS data set to a file.
- PATHOPTS(USE) indicates that TURBO wants to use the PATHOPTS specified on the ALLOCATE command.

For further information about:
- The OPUT and OCOPY commands, see *OS/390 UNIX System Services Command Reference*.
- The ALLOCATE command, see *OS/390 TSO/E Command Reference*.
- The FREE command, see *OS/390 TSO/E Command Reference*.
- The JCL, see *OS/390 MVS JCL Reference*.

Copying a PDS or PDSE to a Directory

The OPUTX command is actually an exec that calls OPUT. You can use the OPUTX command to copy either of these:
- Members of an MVS partitioned data set or PDSE to an HFS directory
- A sequential data set or one member of a partitioned data set to a file

The syntax of the command is:

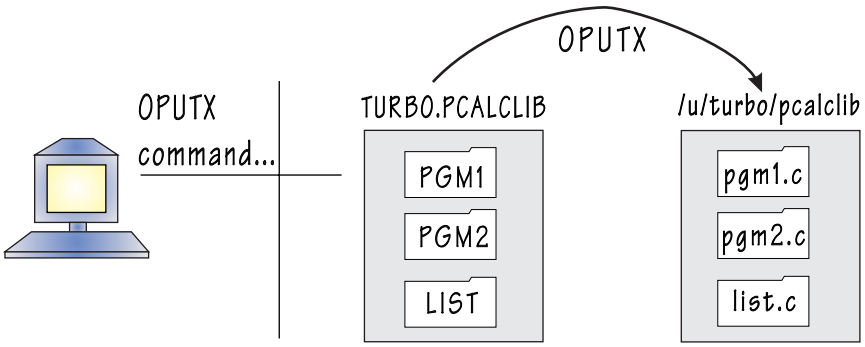
```
OPUTX mvs_PDS_name or mvs_data_set_name[(member_name)]
      'hfs_directory' or 'hfs_file_name'
      {ASIS}
      {TEXT | BINARY}
      {CONVERT(convert_table_name | YES | NO)}
      {LC}
      {MODE}
      {QUIET}
      {SUFFIX{(suffix)}}
```

For the copy, you can specify if this is text or binary data or select code page conversion. When copying a partitioned data set or PDSE, you can specify a copy to lowercase filenames and append a suffix to the member names when they become filenames.

You can use the CONVERT option for singlebyte data, but not for doublebyte data. See “Doublebyte Data” on page 279 for information on code page conversion for doublebyte data.

The single quotes around the directory name and filename are optional. Avoid using OPUTX with pathnames containing quotes and spaces. For details on the OPUTX command, see *OS/390 UNIX System Services Command Reference*.

Example: Using OPUTX with a PDSE



User TURBO wants to copy the members from the data set TURBO.PCALCLIB into the directory **/u/turbo/pcalclib**. He issues the command:

```
OPUTX PCALCLIB /u/turbo/pcalclib LC CONVERT(YES) SUFFIX(c)
```

This command:

- Copies the partitioned data set to a directory. Because the data set name is not enclosed in single quotes, the system automatically uses the data set whose high-level qualifier is the user's user ID.
- Converts data set member names to lowercase filenames.
- Converts the file to code page IBM-1047.
- Appends the suffix **.c** to each filename.

The members of the partitioned data set become files in the directory:

Member name	Filename
TURBO.PCALCLIB(PGM1)	**/u/turbo/pcalclib/pgm1.c**
TURBO.PCALCLIB(PGM2)	**/u/turbo/pcalclib/pgm2.c**
TURBO.PCALCLIB(LIST)	**/u/turbo/pcalclib/list.c**

Copying an MVS VSAM Data Set to an HFS File

To copy a VSAM data set:

1. Use the access method services (AMS) utility to move the VSAM data set to a sequential data set.
2. Copy the MVS sequential set to an HFS file. See "Copying a Sequential Data Set or PDS Member into an HFS File" on page 280 for instructions.

To move the VSAM data set to an HFS file permanently, delete the data set from MVS with the TSO/E DELETE command.

Copying an HFS File into a Sequential Data Set or PDS Member

You might want to copy an HFS file to a sequential data set or to a member of a partitioned data set or PDSE. After it is moved, the file:

- Can be data for an existing MVS application program.
- Can be sent to another system, including a workstation.

You can copy text files or binary files. See “Copying an Executable Module from the File System” on page 294 for more information about copying an executable.

If the HFS files are no longer needed after they have been copied, remove them from their HFS directory.

To copy the data, use the TSO/E OGET command or the TSO/E OCOPY command. You can enter either command:

- In TSO/E, in the shell, or in ISPF. See “Entering a TSO/E Command” on page 202 for information on entering TSO/E commands in TSO/E, the shell, and ISPF.
- In batch, using a Terminal Monitor Program (TMP) job.

To work with data set names and filenames, use the OGET command. To work with ddnames, use the OCOPY command.

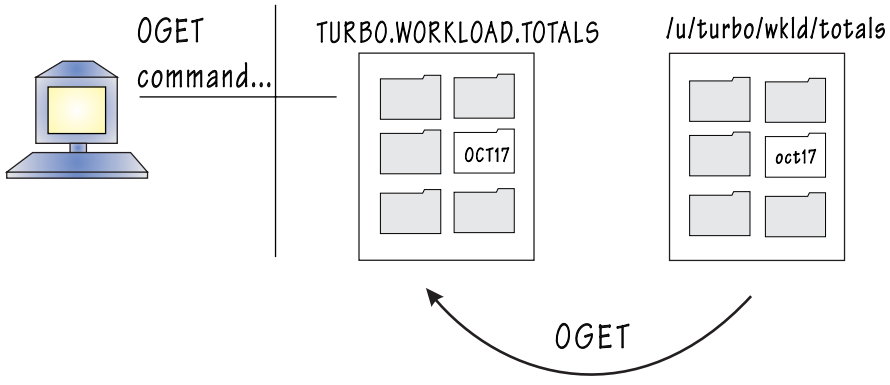
OGET

The OGET command syntax is:

```
OGET 'hfs_file_name'
     mvs_data_set_name[(member_name)]
     {TEXT | BINARY}
     {CONVERT(convert_table_name | YES | NO)}
```

You can use the CONVERT option for singlebyte data, but not for doublebyte data. See “Doublebyte Data” on page 279 for information on code page conversion for doublebyte data.

Example: Using OGET with a PDSE member



If a person with the user ID TURBO enters the following command:

```
OGET '/u/turbo/wkld/totals/oct17' WORKLOAD.TOTALS(OCT17) CONVERT(YES)
```

the system:

- Copies the text file **/u/turbo/wkld/totals/oct17** into the member OCT17 of the PDSE TURBO.WORKLOAD.TOTALS. (The default file type is a text file.)
- Converts the data from code page IBM-1047 to the MVS code page IBM-037, using the default conversion table. You can specify a table name if you do not want to use the default table. If you do not want conversion, omit the CONVERT operand.

  For more information, see "Copying Data: Code Page Conversion" on page 279.

If a member by this name already exists in the data set, it is replaced. If the member does not exist, a new member is created. However, if a partitioned data set or PDSE does not exist, it is not allocated.

If you are moving the HFS file permanently to an MVS data set, remove it from the file system with the **rm** shell command.

Example: Using OGET with a Sequential Data Set

If a person with the user ID TURBO enters the following command:

```
OGET '/u/turbo/wkld/proja/nov' WORKLOAD.PROJA.NOV CONVERT(YES)
```

the system:

- Copies the text file **/u/turbo/wkld/proja/nov** into the sequential data set TURBO.WORKLOAD.PROJA.NOV. (The default file type is a text file.)
- Converts the data from code page IBM-1047 to the MVS code page IBM-037, using the default conversion table. You can specify a table name if you do not want to use the default table. If you do not want conversion, omit the CONVERT operand.

  For more information, see "Copying Data: Code Page Conversion" on page 279.

If a data set with this name already exists, it is replaced. If the sequential data set does not exist, it is automatically allocated. For details on the format and size of the data set that is allocated, see the OGET command description in *OS/390 UNIX System Services Command Reference*.

If you are moving the HFS file permanently to an MVS data set, remove it from the file system with the **rm** shell command.

OCOPY

To copy an HFS file into an MVS data set using data definition names (ddname) instead of a data set name or pathname, use the OCOPY command:

```
OCOPY INDD(ddname1) OUTDD(ddname2)
      {TEXT | BINARY}
      {CONVERT(convert_table_name | YES | NO)}
      {TO1047 | FROM1047}
      {PATHOPTS(USE | OVERRIDE)}
```

1. If the HFS file and data set are not yet allocated, allocate them and specify ddnames, either using the TSO/E ALLOCATE command or the DD statement for JCL.

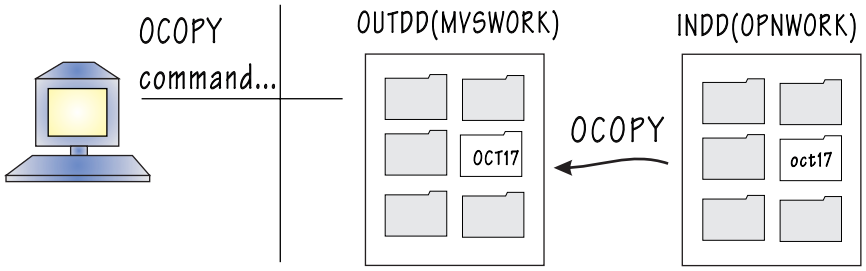
   The ALLOCATE command has these keywords for allocating an HFS file:

   ```
   PATH
   PATHDISP
   PATHMODE
   PATHOPTS
   ```

They are explained in *OS/390 TSO/E Command Reference*.

2. Enter the OCOPY command, being sure the ddnames used match the ddnames specified when the data set and file were allocated.
3. You can use the CONVERT option for singlebyte data, but not for doublebyte data. See "Doublebyte Data" on page 279 for information on code page conversion for doublebyte data.
4. After the copy is completed, you can delete the HFS file using the **rm** shell command.

Example: Using ALLOCATE and OCOPY



1. Using the ALLOCATE command to associate an existing HFS file with the ddname specified in the DDNAME keyword, user TURBO could enter:

```
ALLOCATE DDNAME(OPNWORK) PATH('/u/turbo/wkld/totals/oct17')
         PATHOPTS(ORDWR,OAPPEND) PATHDISP(KEEP,KEEP)
```

   In this example:

   - The file already exists, and PATHOPTS(ORDWR,OAPPEND) indicates that the file owner has read/write access to the file and the owner's data should be written at the end of the file.
   - The PATHDISP(KEEP,KEEP) indicates the file saved in case of normal or abnormal termination.

2. Using the ALLOCATE command to associate the output data set with the ddname specified in the DDNAME keyword, user TURBO could enter:

```
ALLOCATE DDNAME(MVSWORK) DSNAME('TURBO.WORKLOAD.TOTALS(OCT17)') OLD
```

   where the DDNAME keyword specifies the ddname. OLD indicates this is an existing data set and others cannot access the data set while the system is writing to it.

   Note: For an ALLOCATE, you can enter the data set name more simply as `DSNAME(WORKLOAD.TOTALS(OCT17))`—without the user ID. (TSO/E automatically prefixes the data set name with your user ID if you do not enclose the name in quotes.) For JCL, you need the user ID.

3. Then TURBO enters the OCOPY command, using ddnames, to copy the HFS file to an MVS data set:

```
OCOPY INDD(OPNWORK) OUTDD(MVSWORK) TEXT CONVERT(YES) PATHOPTS(USE)
```

   PATHOPTS(USE) indicates that TURBO wants to use the PATHOPTS specified on the ALLOCATE command.

Example: Using JCL and OCOPY

Alternatively, TURBO could specify the ddnames in the //IN DD and //OUT DD statements in the JCL for a batch job. A DD statement allocates a data set or file and sets up a ddname. For example:

```
//TEST JOB MSGLEVEL=(1,1)
//COPYSTEP EXEC PGM=IKJEFT01
//INHFS    DD PATH='/u/turbo/wkld/totals/oct17',PATHOPTS=(ORDONLY)
//OUTMVS   DD DSN=TURBO.WORKLOAD.TOTALS(OCT17),DISP=OLD
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
OCOPY INDD(INHFS) OUTDD(OUTMVS) TEXT CONVERT(YES) PATHOPTS(USE)
/*
```

In this example:

- IKJEFT01 is the name of the Terminal Monitor Program (TMP), which needs to be started to process the TSO/E OCOPY command.
- PATHOPTS(USE) indicates that TURBO wants to use the PATHOPTS specified on the ALLOCATE command.

For further information about:

- The OGET and OCOPY commands, see *OS/390 UNIX System Services Command Reference*.
- The ALLOCATE command, see *OS/390 TSO/E Command Reference*.
- The JCL, see *OS/390 MVS JCL Reference*.

Copying a Directory into a PDS or PDSE

The OGETX command is actually an exec that calls OGET. You can use the OGETX command to copy either of these:

- Files from an HFS directory to an MVS partitioned data set or PDSE
- An individual file to a sequential data set or member of a partitioned data set

The syntax of the command is:

```
OGETX 'hfs_directory' or 'hfs_file_name'
      mvs_PDS_name or mvs_data_set_name[(member_name)]
      {ASIS}
      {TEXT | BINARY}
      {CONVERT(convert_table_name | YES | NO)}
      {LC}
      {QUIET}
      {SUFFIX{(suffix)}}
```

For the copy, you can specify text or binary data and select code page conversion. When copying a directory, you can specify a copy from lowercase filenames and delete one or all suffixes from the filenames when they become PDS member names. For a file to be copied, its name must conform to partitioned data set member name conventions after any suffix and LC processing is done. Member names can be 1–8-character uppercase alphanumeric or national characters (A–Z, 0–9, $, #, @). They cannot start with a numeric.

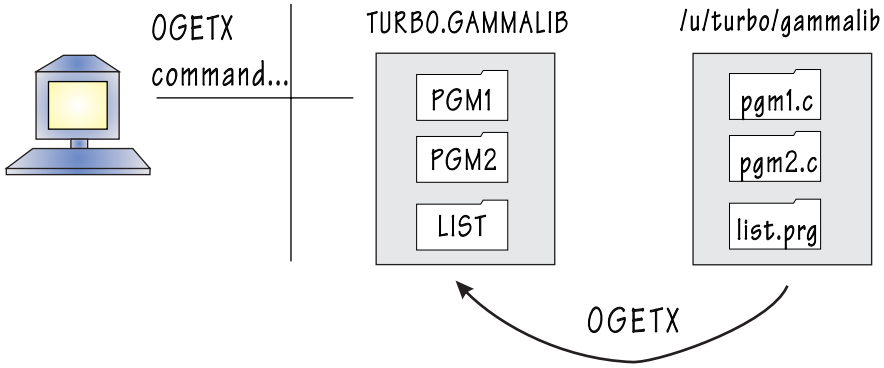
If you specify a particular suffix, only files with that suffix are copied—with the suffix deleted. If you use the SUFFIX operand without specifying a particular suffix, any filenames with suffixes have the suffix deleted and all files are copied. (After the suffix is deleted, if more than one file has the same name, each subsequent file copied overlays a file with the same name that was copied previously.)

The single quotes around the directory name and filename are optional. Avoid using OGETX with pathnames containing quotes and spaces.

You can use the CONVERT option for singlebyte data, but not for doublebyte data. See “Doublebyte Data” on page 279 for information on code page conversion for doublebyte data.

If the OGETX command creates a new data set, it has the same format and size as a data set created by the OGET command. For details on the OGETX command, see *OS/390 UNIX System Services Command Reference*.

Example: Using OGETX with a PDSE



User TURBO wants to copy the directory **/u/turbo/gammalib** into the partitioned data set TURBO.GAMMALIB. He issues the command:

```
OGETX /u/turbo/gammalib GAMMALIB LC SUFFIX
```

This command:

- Copies into the partitioned data set all the files in the directory that meet MVS member name requirements. Because the data set name is not enclosed in single quotes, the system automatically supplies the user's user ID (TURBO) as a high-level qualifier.
- Copies from files with lowercase, uppercase, or mixed-case names.
- Removes any suffixes to the filenames. (After the suffix is deleted, if more than one file has the same name, each subsequent file copied overlays a file with the same name that was copied previously.)

The files in the directory become partitioned data set members:

Filename	**Member name**
/u/turbo/gammalib/pgm1.c	TURBO.GAMMALIB(PGM1)
/u/turbo/gammalib/pgm2.c	TURBO.GAMMALIB(PGM2)
/u/turbo/gammalib/list.prg	TURBO.GAMMALIB(LIST)

Copying an HFS File to Another HFS File

You can use the shell command **cp** or the TSO/E command OCOPY to copy files within the file system.

Using the Shell:

Use the **cp** shell command to copy:
- One file to another file in the working directory
- One file to a new file on another directory
- A set of directories and files to another place in your file system

cp copies one or more files to a new location.

```
cp file1 file2
```

copies the contents of **file1** into **file2**. To copy a list of files to the specified directory, enter:

```
cp file1 file2 file3 ... directory
```

For example:

```
cp dir1/a dir2/b dir3
```

copies two files into the directory called **dir3**. The copied files have same the filenames as the original, so you will find files **a** and **b** in the directory **dir3**.

For further information on the **cp** command, see *OS/390 UNIX System Services Command Reference*.

Using TSO/E:

You can use the TSO/E OCOPY command to copy an HFS file to another HFS file and in the process convert the data from one code page to another.

Here is an example of using the OCOPY command to copy an HFS file to another HFS file in a different directory:

```
ALLOCATE DDNAME(KPAYR) PATH('/u/kinn/bin/payroll')
ALLOCATE DDNAME(MPAYR) PATH('/u/mills/bin/payroll')
OCOPY INDD(KPAYR) OUTDD(mpayr) TEXT CONVERT((BPXFX311)) TO1047
```

The combination of CONVERT((BPXFX311)) and TO1047 indicates that you want to use the ASCII conversion table to convert from ASCII to code page IBM-1047. TO1047 or FROM1047 is required if CONVERT is specified.

With the CONVERT parameter, you can specify a data set name, a member name, or both. In this example, the use of (( )) and no data set name indicates that you are specifying a member that is a module in the standard search order for MVS.

If the files being allocated are new files, PATHOPTS and PATHMODE operands are required.

Copying an MVS Data Set into Another MVS Data Set

You can use the TSO/E OCOPY command to copy an MVS data set into another data set. It has a CONVERT option that lets you convert between these code pages:

- IBM-037 and IBM-1047
- IBM-037 and ISO8859-1
- code pages in a user-defined conversion table

With the TSO/E OCOPY command, you can copy:

- A sequential data set to a sequential data set
- A sequential data set to a partitioned data set or PDSE member
- A partitioned data set or PDSE member to a partitioned data set or PDSE member
- A partitioned data set or PDSE member to a sequential data set

You can enter the command:

- In TSO/E, in the shell, or in ISPF. See “Entering a TSO/E Command” on page 202 for information on entering TSO/E commands in TSO/E, the shell, and ISPF.
- In batch, using a Terminal Monitor Program (TMP) job.

The OCOPY command uses ddnames instead of data set names:

```
OCOPY INDD(ddname1) OUTDD(ddname2)
      {TEXT | BINARY}
      {CONVERT(convert_table_name | YES | NO)}
      {TO1047 | FROM1047}
```

You do not need the PATHOPTS operand when copying from one data set to another.

There are two ways to specify ddnames, either using the ALLOCATE command or JCL for a batch job.

Example: Using ALLOCATE and OCOPY

Using the ALLOCATE command to associate each data set with a ddname, user TURBO could enter:

```
ALLOCATE DDNAME(TMP1) DSNAME(TEMP1) SHR
ALLOCATE DDNAME(TMP1OC) DSNAME(TEMP1OC) OLD
```

where the DDNAME keyword specifies the ddname. SHR indicates that this is an existing data set and others can access it while the system is reading from it. OLD indicates that this is an existing data set and others cannot access the data set while the system is writing to it.

Then TURBO could enter the OCOPY command, using the ddnames from the ALLOCATE command, to convert the data in TEMP1 from the MVS country-extended code page to code page IBM-1047 and copy it to the data set TEMP1OC:

```
OCOPY INDD(TMP1) OUTDD(TMP1OC) TEXT CONVERT(YES) TO1047
```

If CONVERT is specified, you must also specify either TO1047 or FROM1047.

Example: Using JCL and OCOPY

Alternatively, TURBO could specify the ddnames in the //IN DD and //OUT DD statements in the JCL for a batch job. A DD statement allocates a data set or file and sets up a ddname. For example:

```
//TEST JOB MSGLEVEL=(1,1)
//COPYSTEP EXEC PGM=IKJEFT01
//IN       DD DSN=TURBO.TEMP1,DISP=SHR
//OUT      DD DSN=TURBO.TEMP1OC,DISP=OLD
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
OCOPY INDD(IN) OUTDD(OUT) TEXT CONVERT(YES) TO1047
/*
```

In this example,

- IKJEFT01 is the name of the Terminal Monitor Program (TMP), which needs to be started to process the TSO/E OCOPY command.
- TO1047 is required because you are copying from one data set to another data set.

Copying Executable Modules between MVS and the File System

Suppose you have a program you compiled in MVS and you want to copy it to the file system. The method for copying an executable load module; from an MVS data set to the file system depends on the kind of data set the module is in: PDSE or partitioned data set.

To copy an executable from the file system into an MVS PDS or PDSE, you can use OGETX.

Copying an Executable Module from a PDSE

If the load module is in a PDSE, you can copy it to the file system, using one of the following commands:

- The OPUT or OPUTX command. For a new text (non-U-format data set) file, the default permission is octal 600; you can use the **chmod** command or the MODE keyword on the OPUTX command to make it executable. If you replace an existing file, the existing permissions are unchanged.
- The OCOPY command. Specify PATHMODE(SIRWXU) to make the file executable for the owner.

Copying an Executable Module from a PDS

If the load module is in a partitioned data set (PDS), you can do one of these:

- Use OPUTX. If the source data set is a PDS with an undefined record format, OPUTX treats the members as load modules. In order for the program to be able to run from the file hierarchy, the entry point must be at the beginning of the load module.

  In order for OPUTX to treat the file as a load module, do not specify BINARY or TEXT. Once the module is in the file system, use **chmod** to make it executable. If you replace an existing file, the permissions are unchanged.
- Use JCL that invokes the binder before copying the module into the file system. See the following example for sample JCL.

Example: Using JCL to Copy from a PDS to the File System

To copy a load module out of a partitioned data set and into the file system, you have to use the binder to “flatten” the load module. Here is an example of JCL our friend TURBO wrote for copying a OS/390 c/c++ load module into the file system:

```
//TURBO     JOB (XX,YY,ZZ),MSGCLASS=H,CLASS=A,
//          MSGLEVEL=(1,1)
//*
//LKED      EXEC PGM=IEWBLINK,REGION=500K,
//          PARM='LIST,REUS,RENT,NCAL,LET,MAP,CASE=MIXED'
//SYSPRINT DD  SYSOUT=*
//INLIB    DD  DSN=TURBO.POSIX.LOADLIB,DISP=SHR
//*
//SYSLMOD  DD  PATH='/u/turbo/llib',
//             PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//             PATHMODE=(SIRWXO,SIRWXG,SIRWXU)
//*
//SYSLIN   DD  *
        INCLUDE INLIB(PAYRLL)
        ENTRY CEESTART
        NAME payrll(R)
/*
```

This job relinks, or rebinds, the OS/390 c/c++ load module, PAYRLL, from TURBO.POSIX.LOADLIB(PAYRLL) and puts the output into the file system as **/u/turbo/llib/payrll**. Be sure you specify the correct entry point—in this case, CEESTART—for a OS/390 c/c++ program. If you do not specify the entry point, the entry point is assumed to be at the beginning of the load module.

Copying an Executable Module from the File System

There are two methods for *copying* an executable module from the file system into a data set. However, these methods are not exactly copy operations, but instead, they bind the executables over to the data set. As a result, certain attributes are not preserved, but rather, re-established:

- Use OGETX without the TEXT or BINARY option. If you are copying into a target data set that is a PDS or PDSE with an undefined record format, OGETX treats the files as executables. Since the entry point is re-established, this option only works if the original entry point is at the beginning of the executable.

  Along with the entry point, other attributes are re-established such as the authorization (AC) value, which is reset to AC=1 (the AUTH option allows it to be set to AC=1).

  Be aware that most executables created with OS/390 V2R4 or later will not copy successfully into a PDS. When you attempt to execute this kind of operation, you’ll receive the following type of error:

  ```
  IEW2606S 4B39 MODULE INCORPORATES PROGRAM MANAGEMENT 3
  FEATURES AND CANNOT BE SAVED IN LOAD MODULE FORMAT.
  ```

  Instead of copying into a PDS, executables created with OS/390 V2R4 or later can be copied into a PDSE.

  Another exception are DLL-enabled executables created with OS/390 V2R4 or later. These executables will not copy successfully into both PDSs and PDSEs. This occurs because the information provided by the IMPORT control statements is not preserved and must be specified again during the rebind.
- Use JCL that invokes the binder before copying the executable into the data set. See the following example for sample JCL.

Example: Using JCL to Copy from the File System to a PDS

To copy an executable out of the file system and into a data set, you need to use the binder to reprocess the executable. Here is an example JCL our friend TURBO wrote for copying an OS/390 c/c++ load module into a data set:

```
//TURBO     JOB (XX,YY,ZZ),MSGCLASS=H,CLASS=A,
//          MSGLEVEL=(1,1)
//*
//LKED      EXEC PGM=IEWBLINK,REGION=500K,
//          PARM='LIST,REUS,RENT,NCAL,LET,MAP'
//SYSPRINT DD  SYSOUT=*
//SYSLMOD  DD  DSN=TURBO.POSIX.LOADLIB,DISP=SHR
//*
//INLIB    DD  PATH='/u/turbo/llib/payrll',
//             PATHOPTS=(ORDONLY)
//*
//SYSLIN   DD  *
         INCLUDE INLIB
         ENTRY CEESTART
         NAME PAYRLL(R)
/*
```

This job relinks, or rebinds, the OS/390 c/c++ executable **/u/turbo/llib/payrll** and puts the output into TURBO.POSIX.LOADLIB(PAYRLL). Be sure you specify the correct entry point—in this case, CEESTART—for a OS/390 c/c++ program. If you do not specify the entry point, the entry point is assumed to be at the beginning of the executable. Also, if required, you must specify AC=1.

If this is a DLL created using V2R4 or later and without the use of the prelinker, then any definition side-decks of IMPORT control statements will need to be re-specified as input to the binder. In general, any control statements and options used when the original **/u/turbo/llib/payrll** was created will need to be specified again.

Chapter 21. Transferring Files between Systems

You may typically create applications and files at your workstation and then move the resulting files to the hierarchical file system (HFS) for further application development—such as compiling and debugging, or to share the files. There may also be times when you want to send HFS files to your workstation. The following text discusses several methods for moving files directly between your workstation and the HFS.

File Transfer Directly to/from the HFS

To move a file or file system between your workstation and the HFS, you can use one of the following methods.

Transferring Files Using File Transfer Protocol (FTP)

You can use the File Transfer Protocol (FTP) facility of TCP/IP, when both the workstation and MVS system have TCP/IP installed.

With the OS/390 Communications Server installed on a remote OS/390 system, you can **ftp** files into or from that system's HFS.

An FTP client is not available for the shell and utilities. However, we have ported **ncftp**, an FTP client, and it is available at `http://www.s390.ibm.com/unix/bpxa1toy.html` for downloading. You can use it to **ftp** files into or from a local HFS.

Transferring Files Using the Network File System Feature

Using the Network File System feature, you can edit or browse an HFS file directly from your workstation. If you want to copy an HFS file to a workstation file, you do not need to move it to an MVS data set first. Here is an example showing the series of steps involved:

1. You log on to the host using **mvslogin**.
2. You mount the HFS directory **/u/usr1/a/b** at the workstation with the command:

   ```
   mount mvshost:"/u/usr1/a/b" /x/y
   ```
3. You copy the HFS file **/u/usr1/a/b/c** to the workstation file **/mycopy/c** with the command:

   ```
   cp /x/y/c /mycopy/c
   ```

Using the Network File System feature from your workstation, you can copy a workstation file to an HFS file without having to move it to an MVS data set first. This example assumes you have run your **mvslogin** and mounted the HFS directory **/u/usr1/pgma/b** at the workstation under the pathname **/mypgma/b**. You copy the workstation file **/proj2/modc** to the HFS file **/u/usr1/pgma/b/modc** with the command:

```
cp /proj2/modc /mypgma/b/modc
```

Suppose you have an executable that you compiled and linked at a workstation, and you want to store it in an MVS HFS but run it from the workstation. You copy the executable to the mounted HFS in binary format. Later, when you want to run the program from the workstation, you use NFS to mount the HFS directory in binary format, and then run the program from the mounted HFS.

For more information about working with HFS files at your workstation, see *DFSMS/MVS: Network File System User's Guide*.

Transferring Files Using the SEND and RECEIVE Programs

The SEND and RECEIVE programsavailable with PC 3270 emulation programs and with OS/2 Extended Edition Version 1.2 or later.

Note: Before using the SEND and RECEIVE programs, you must be working in TSO/E. If you are using the OMVS interface to work in the shell, use the TSO function key to switch to TSO/E command mode *before* using the programs.

Transferring Files Using the File Transfer, Access, and Management Function

You can also transfer files between your workstation and the HFS using the File Transfer, Access, and Management (FTAM) function of OSI/File Services.

File Transfer Using MVS Data Sets

Transferring files between systems can also take place without the Network File System feature.

Transferring Files into the HFS

If the OS/390 Communications Server is installed on a remote system, you can **ftp** files directly into that system's HFS.

Note: If you are ftp-ing to a remote OS/390 HFS, be aware that the OS/390 UNIX server often listens to a port other than the well-known port. Make sure you know the address and port to use.

If you are not using the Network File System feature and the OS/390 Communications Server is not installed, perform these steps:

1. Transfer the data to the host, using your preferred method, for example, FTP.
2. When logged on to TSO/E, copy the data from an MVS data set into the file system, using the TSO/E OPUT command.

   Singlebyte data: If you need to convert to a shell-supported code page, use the CONVERT option on the OPUT command. See "OPUT" on page 280.

   Doublebyte data or multibyte ASCII-based data: If you need to convert to a shell-supported code page, use either the OS/390 c/c++**iconv** utility while working in MVS, or the **iconv** shell utility while working in the shell. For more information, see "Copying Data: Code Page Conversion" on page 279.
3. If desired, after the copy you can delete the MVS data set with the TSO/E DELETE command.

Transferring Files to the Workstation

If you are working without the Network File System feature, perform these steps when logged on to TSO/E:

1. Copy the HFS file to an MVS data set (sequential or partitioned) using the TSO/E OGET command. See "OGET" on page 286.

   Singlebyte data: If you need to convert to a different code page, you can use the CONVERT option on the OGET command.

Doublebyte data: If you need to convert the data, you can use the **iconv** utility while working in the shell. For more information, see “Copying Data: Code Page Conversion” on page 279.

2. If desired, after the copy you can delete the HFS file with the **rm** shell command.
3. Send the data set to the workstation, using your preferred method, for example, FTP.

Transporting an Archive File on Tape or Diskette

A directory or file system that is going to be transported on tape or diskette is put into an archive file, as discussed in “Backing Up and Restoring Files: The Options” on page 228. This section discusses the steps involved in

- Installing an archive file from tape or diskette into an HFS file system
- Putting an archive file on tape or diskette to send to another site

Putting an Archive File into the File System

You may receive an archive file on tape or diskette. There are two major steps involved in installing the archive file in an HFS file system:

1. Transferring the archive file into an MVS data set, from either:
   - A workstation
   - A tape drive at your MVS system
2. Copying the archive file from the data set into the file system

Step 1. Transferring the Archive File to a Data Set
From a Workstation:

If you have TCP/IP on your workstation, you can use the **ftp** command to transfer an archive file to MVS or to the OS/390 shell (if you have the OS/390 Communications Server installed).

At the workstation:

a. Copy the archive file into a file from one of these:
- A diskette for an RS/6000
- A tape for an RS/6000

b. Enter the **FTP** command.

Note: If you are ftp-ing to a remote OS/390 HFS, be aware that the OS/390 UNIX server often listens to a port other than the well-known port. Make sure you know the address and port to use.

c. Enter the **binary** subcommand.

d. Enter the **put** subcommand, specifying an HFS directory or a sequential or partitioned data set as the destination.

If you are specifying a data set, you may prefer to use one partitioned data set for all your archive files, with each archive file a member in the partitioned data set. Here is an example of the partitioned data set attributes you might want:

```
DATA SET NAME: TURBO.CMPL.ARCHIVE

GENERAL DATA:                              CURRENT ALLOCATION:
   Volume serial:         TRBLK1             Allocated Cylinders: 26
   Device type:           3380               Allocated extents:    5
   Organization:          PO
   Record format:         VB
   Record length:         255
   Block size:            23476           CURRENT UTILIZATION:
   1st extent Cylinders:  12                 Used Cylinders:       0
   Secondary Cylinders:   0                  Used extents:         0

   Creation date:         1994/12/18
   Expiration date:       ***NONE***
```

e. Go to "Step 2. Copying the File from a Data Set into a File System".

From a Tape Drive for Your MVS System:

If you have an archive file on tape and the necessary tape drive at your MVS system, you can copy the file directly from the tape into a data set.

Working at MVS:

a. Copy the archive file from the tape into a data set. Here is some sample JCL for copying an archive file (TURBO.TARTAPE) from a tape into a data set (TURBO.TAR):

```
//TAPE2DS   JOB ,',MSGLEVEL=(1,1)
//*
//STEP1    EXEC PGM=IEBGENER
//SYSIN    DD  DUMMY
//SYSPRINT DD  SYSOUT=*
//SYSUT1   DD  DSNAME=TURBO.TARTAPE,UNIT=TAPE,LABEL=(1,NL),DISP=OLD,
//             VOL=SER=123456,DCB=(RECFM=U,BLKSIZE=5120)
//SYSUT2   DD  DSNAME=TURBO.TAR,DISP=(NEW,CATLG),UNIT=SYSDA,
//             SPACE=(5120,(100,100),RLSE)
```

Note: For LABEL=, NL indicates that there is no label. Use NL when transferring a tape between an MVS and a UNIX system; use SL when transferring a tape between two MVS systems.

b. Go to Step 2. Copying the File from a Data Set into a File System, which follows.

Step 2. Copying the File from a Data Set into a File System

Working at MVS:

Note: With Release 8 and later, the **pax** and **tar** utilities can read an archive file directly from an MVS dataset. If you are using Release 8, skip step a.

a. With Release 7 or earlier, use the TSO/E OPUT command with the BINARY option to copy the partitioned data set member or sequential data set into the file system. See "OPUT" on page 280 for more information. The archived file becomes a single file in the file system.

b. Use the **pax**, or **tar** shell command to restore the directory or file system from the archive file; all the component files are restored from the archive file.

If you need to convert the source to the code page IBM-1047 used in the OS/390 shells, use the **pax** command with the **–o** option. See “Backing Up and Restoring Files: The Options” on page 228 for more information.

Sending an Archive File to Others

The following shows the steps to send an archive file that contains multiple files on tape or diskette. In the example, the **pax** command creates an archive file for a directory or file system. Then the TSO/E OGET command, with the BINARY option, copies the archive file into a partitioned data set or a sequential data set. Step 2 is not necessary with Release 8.

Step 1. Create an Archive File for Multiple Files

You can use either the **pax**, or **tar** shell command to create the archive file. All the component files are stored in one archive file. For example, to archive all subdirectories and files in **/tmp/posix/testpgm** to the archive **testpgm.pax**:

```
pax -wf /tmp/testpgm.pax -o from=IBM-1047,to=ISO8859-1 /tmp/posix/testpgm
```

For the **pax** command:

- The **—w** option writes to the archive file.
- The **—f** option lets you specify the name of the archive file.

If you need to convert to a different code page than the one used in the shell, use the **pax** command with the **–o** option. See “Backing Up and Restoring Files: The Options” on page 228 for more information.

Step 2. Copy the File from the File System to a Data Set

Use the TSO/E OGET command with the BINARY option to copy the archive file into a sequential data set. See “OGET” on page 286 for more information.

```
tso "OGET '/tmp/testpgm.pax' 'POSIX.TESTPGM.PAX' BINARY"
```

The OGET command copies the archive file into the specified MVS data set:

- **'/tmp/posix/testpgm.pax'** is the absolute pathname for the archive file.
- **'POSIX.TESTPGM.PAX'** is the fully qualified data set name for the data set.
- BINARY indicates the data is binary.

The final step is to use **ftp**, or some other method, to send the file to the intended destination.

Step 3. Transfer the Archive File to a Tape or Diskette

To a Tape or Diskette at the Workstation:

Working at MVS:

a. For information on how to copy an archive file from the file system into a data set, see “Step 2. Copy the File from the File System to a Data Set”.

b. Enter the **FTP** command.

c. Enter the **binary** subcommand.

d. Enter the **put** subcommand, specifying a pathname at your workstation as a destination.

e. At the workstation, copy the archive file onto one of these:
- A diskette, for an RS/6000
- A tape, for an RS/6000

To a Tape at the Host:

Working at MVS:

a. For information on how to copy an archive file from the file system into a data set, see “Step 2. Copy the File from the File System to a Data Set” on page 301.

b. Copy the archive file from the data set to tape. Here is some sample JCL for copying a data set containing an archive file (TURBO.TAR) to a tape (TURBO.TARTAPE):

```
//DS2TAPE  JOB ,',MSGLEVEL=(1,1)
//*
//STEP1    EXEC PGM=IEBGENER
//SYSIN    DD  DUMMY
//SYSPRINT DD  SYSOUT=*
//SYSUT1   DD  DSNAME=TURBO.TAR,DISP=OLD
//SYSUT2   DD  DSNAME=TURBO.TARTAPE,UNIT=TAPE,LABEL=(1,NL),
//             DISP=(NEW,KEEP)
```

Note: For LABEL=, NL indicates that there is no label. Use NL when transferring a tape between an MVS and a UNIX system; use SL when transferring a tape between two MVS systems.

Part 3. Appendixes

Appendix A. OS/390 Shell Command Summary

The following list presents OS/390 shell commands and utilities grouped by the task a user might want to perform. Similar tasks are organized together. Stub commands (**cancel**, **cu** and **lpstat**) are not listed because their functions are not supported by OS/390 UNIX System Services.

The list also shows the command name, the standard or specification it satisfies, and its function. XPG4.2 refers to "X/Open CAE Issue 4 Version 2 Specifications". XPG5.0 refers to "X/Open CAE Issue 5 Specifications".

General Use

at	POSIX.2	XPG4.2	Run a command at a specified time
batch	POSIX.2	XPG4.2	Run commands when the system is not busy
bpxmtext	—	—	Display reason code text
command	POSIX.2	XPG4.2	Run a simple command
confighfs	—	—	Invoke vfs_pfsctl HFS functions
date	POSIX.2	XPG4.2	Display the date and time
echo	POSIX.2	XPG4.2	Write arguments to standard output
exec	POSIX.2	XPG4.2	Run a command and open, close, or copy the file descriptors
man	POSIX.2	XPG4.2	Print sections of the online reference manual
nice	POSIX.2	XPG4.2	Run a command at a different priority
passwd	—	—	Change user passwords
print	—	—	Return arguments from the shell
printf	POSIX.2	XPG4.2	Write formatted output
sh	POSIX.2	XPG4.2	Invoke a shell (command interpreter)
tcsh	—	—	Invoke a tcsh shell
time	POSIX.2	XPG4.2	Display processor and elapsed times for a command
wall	—	—	Broadcast a message to logged-in users
whence	—	—	Tell how the shell interprets a command name
whoami	—	—	Display your effective username

Controlling Your Environment

alias	POSIX.2	XPG4.2	Display or create a command alias
asa	POSIX.2	XPG4.2	Interpret ASA/Fortran carriage control
automount	—	—	Configure the automount facility
cal	—	XPG4.2	Display a calendar for a month or year
calendar	—	XPG4.2	Display all current appointments
captoinfo	—	—	Prints terminal entries in the termcap file
chcp	—	—	Set or query ASCII/EBCDIC code pages for the terminal
configstk	—	—	Configure the AF_UENT stack
env	POSIX.2	XPG4.2	Display environments, or set an environment for a process
export	POSIX.2	XPG4.2	Set the export attributes for variables, or show currently exported variables
fc	POSIX.2	XPG4.2	Process a command history list
hash	—	XPG4.2	Create a tracked alias

history	—	—	Process a command history list
id	POSIX.2	XPG4.2	Return the user identity
infocmp	—	—	Compare and print the terminal description
ipcrm	—	—	Remove message queue, semaphore set, or shared memory identifiers
ipcs	—	—	Report status of the interprocess communication facility
locale	POSIX.2	XPG4.2	Get locale-specific information
localedef	POSIX.2	XPG4.2	Define the locale environment
logger	POSIX.2	XPG4.2	Log messages
logname	POSIX.2	XPG4.2	Return a user's login name
newgrp	POSIX.2	XPG4.2	Change to a new group
printenv	—	—	Display the value of environment variables
r	—	—	Process a command history list
readonly	POSIX.2	—	Mark a variable as read-only
return	POSIX.2	XPG4.2	Return from a shell function or **.** (dot) script
set	POSIX.2	XPG4.2	Set or unset command options and positional parameters
shift	POSIX.2	XPG4.2	Shift positional parameters
stty	POSIX.2	XPG4.2	Set or display terminal options
su	—	—	Change the user ID connected with a session
sysvar	—	—	Display static system symbols
tic	—	—	Compile term descriptions into terminfo database entries
touch	POSIX.2	XPG4.2	Change the file access and modification times
tput	POSIX.2	XPG4.2	Change characteristics of terminals
tso	—	—	Run a TSO command from the shell
tty	POSIX.2	—	Return the user's terminal name
uconvdef	—	—	Create binary conversion tables
unalias	POSIX.2	XPG4.2	Remove alias definitions
uname	POSIX.2	XPG4.2	Display the name of the current operating system
unset	POSIX.2	XPG4.2	Unset values and attributes of variables and functions
who	POSIX.2	XPG4.2	Display information about current users

Daemons

cron	—	—	Run commands at specified dates and times
inetd	—	—	Handle login requests
rlogind	—	—	Validate rlogin requests
uupd	—	—	Invoke **uucico** for TCP/IP connections from remote UUCP systems

Managing Directories

basename	POSIX.2	XPG4.2	Return the nondirectory components of a pathname
cd	POSIX.2	XPG4.2	Change the working directory
chgrp	POSIX.2	XPG4.2	Change the group owner of a file or directory
chmod	POSIX.2	XPG4.2	Change the mode of a group or directory
chmount	—	—	Change the mount attributes of a file system

chown	POSIX.2	XPG4.2	Change the owner or group of a file or directory
chroot	—	—	Change the root directory for the execution of a command
dircmp	—	XPG4.2	Compare directories
dirname	POSIX.2	XPG4.2	Return the directory components of a pathname
ls	POSIX.2	XPG4.2	List file and directory names and attributes
mkdir	POSIX.2	XPG4.2	Make a directory
mount	—	—	Logically mount a file system
mv	POSIX.2	XPG4.2	Rename or move a file or directory
pathchk	POSIX.2	XPG4.2	Check a pathname
pwd	POSIX.2	XPG4.2	Return the working directory name
rm	POSIX.2	XPG4.2	Remove a directory entry
rmdir	POSIX.2	XPG4.2	Remove a directory
unlink	—	XPG5.0	Removes a directory entry

Managing Files

cat	POSIX.2	XPG4.2	Concatenate or display text files
chaudit	—	—	Change audit flags for a file
cksum	POSIX.2	XPG4.2	Calculate and write checksums and byte counts
cmp	POSIX.2	XPG4.2	Compare two files
col	—	XPG4.2	Remove reverse line feeds
comm	POSIX.2	XPG4.2	Show and select or reject lines common to two files
compress	—	XPG4.2	Lempel-Ziv file compression
cp	POSIX.2	XPG4.2	Copy a file
csplit	POSIX.2	XPG4.2	Split text files
ctags	POSIX.2	XPG4.2	Create tag files for **ex**, **more**, and **vi**
dot or **.**	—	XPG4.2	Run a shell file in the current environment
cut	POSIX.2	XPG4.2	Cut out selected fields of each line of a file
dd	POSIX.2	XPG4.2	Convert and copy a file
df	POSIX.2	XPG4.2	Display the amount of free space in the file system
diff	POSIX.2	XPG4.2	Compare two text files and show the differences
du	POSIX.2	XPG4.2	Summarize usage of file space
ed	POSIX.2	XPG4.2	Use the **ed** line-oriented text editor
egrep	—	XPG4.2	Search a file for a specified pattern
ex	POSIX.2	XPG4.2	Use the **ex** text editor
exrecover	—	—	**vi** file recovery daemon
extattr	—	—	Set, reset, or display extended attributes for files
expand	POSIX.2	XPG4.2	Expand tabs to spaces
fgrep	—	XPG4.2	Search a file for a specified pattern
file	POSIX.2	XPG4.2	Determine file type
filecache	—	—	Manage file caches
find	POSIX.2	XPG4.2	Find a file meeting specified criteria
fold	POSIX.2	XPG4.2	Break lines into shorter lines
head	POSIX.2	XPG4.2	Display the first part of a file
iconv	—	XPG4.2	Convert characters from one code set to another
join	POSIX.2	XPG4.2	Join two sorted, textual relational databases

line	—	XPG4.2	Copy one line of standard input
link	—	XPG5.0	Create a hard link to a file
ln	POSIX.2	XPG4.2	Create a link to a file
mkfifo	POSIX.2	XPG4.2	Make a FIFO special file
mknod	—	—	Make a FIFO or character special file
mount	—	—	Logically mount a file system
more	POSIX.2	XPG4.2	Display files on a page-by-page basis
mv	POSIX.2	XPG4.2	Rename or move a file or directory
nl	—	XPG4.2	Number lines in a file
nm	POSIX.2	XPG4.2	Display symbol table of object, library, and executable files
obrowse	—	—	Browse an HFS file
od	POSIX.2	XPG4.2	Dump a file in a specified format
oedit	—	—	Edit an HFS file
pack	—	XPG4.2	Compress files by Huffman coding
paste	POSIX.2	XPG4.2	Merge corresponding or subsequent lines of a file
patch	POSIX.2	XPG4.2	Change a file using diff output
pcat	—	XPG4.2	Display Huffman-packed lines on standard output
pg	—	XPG4.2	Display files interactively
sed	POSIX.2	XPG4.2	Start the **sed** noninteractive stream editor
skulker	—	—	Delete files over a certain age
sort	POSIX.2	XPG4.2	Start the sort-merge utility
spell	—	XPG4.2	Detect spelling errors in files
split	POSIX.2	XPG4.2	Split a file into manageable pieces
strings	POSIX.2	XPG4.2	Display printable strings in binary files
sum	—	XPG4.2	Compute checksum and block count for file
tabs	POSIX.2	XPG4.2	Set tab stops
tail	POSIX.2	XPG4.2	Display the last part of a file
tee	POSIX.2	XPG4.2	Duplicate the output stream
tr	POSIX.2	XPG4.2	Translate characters
tsort	—	XPG4.2	Sort files topologically
umask	POSIX.2	XPG4.2	Set or return the file mode creation mask
uncompress	—	XPG4.2	Undo Lempel-Zev compression of a file
unexpand	POSIX.2	XPG4.2	Compress spaces into tabs
uniq	POSIX.2	XPG4.2	Report or filter out repeated lines in a file
unmount	—	—	Remove a file system from the file hierarchy
unpack	—	XPG4.2	Decode Huffman packed files
uudecode	POSIX.2	XPG4.2	Decode a transmitted binary file
uuencode	POSIX.2	XPG4.2	Encode a file for safe transmission
vi	POSIX.2	XPG4.2	Use the display-oriented interactive text editor
wc	POSIX.2	XPG4.2	Count newlines, words, and bytes
zcat	—	XPG4.2	Uncompress and display data

Printing Files

cancel	—	—	Cancel print queue requests (stub command)
infocmp	—	—	Compare and print the terminal description
lp	POSIX.2	XPG4.2	Send a file to a printer
lpstat	—	—	Show status of print queues (stub command)
pr	POSIX.2	XPG4.2	Format a file in paginated form and send it to standard output

Computing and Managing Logic

bc	POSIX.2	XPG4.2	Use the arbitrary-precision arithmetic calculation language
break	POSIX.2	XPG4.2	Exit from a for, while, or until loop in a shell script
colon or **:**	POSIX.2	XPG4.2	Do nothing, successfully
continue	POSIX.2	XPG4.2	Skip to the next iteration of a loop in a shell script
dot or **.**	POSIX.2	XPG4.2	Run a shell file in the current environment
eval	POSIX.2	XPG4.2	Construct a command by concatenating arguments
exec	POSIX.2	XPG4.2	Run a command and open, close, or copy the file descriptors
exit	POSIX.2	XPG4.2	Return to the parent process from which the shell was called or to TSO/E
expr	POSIX.2	XPG4.2	Evaluate arguments as an expression
false	POSIX.2	XPG4.2	Return a nonzero exit code
grep	POSIX.2	XPG4.2	Search a file for a specified pattern
left bracket or **[**	—	XPG4.2	Test for a condition
let	—	—	Evaluate an arithmetic expression
test	POSIX.2	XPG4.2	Test for a condition
trap	POSIX.2	XPG4.2	Intercept abnormal conditions and interrupts
true	POSIX.2	XPG4.2	Return a value of `0`

Controlling Processes

bg	POSIX.2	XPG4.2	Move a job to the background
crontab	POSIX.2	XPG4.2	Schedule regular background jobs
fg	POSIX.2	XPG4.2	Bring a job into the foreground
fuser	—	XPG5.0	List process IDs of processes with open files
jobs	POSIX.2	XPG4.2	Return the status of jobs in the current session
kill	POSIX.2	XPG4.2	End a process or job, or send it a signal
nohup	POSIX.2	XPG4.2	Start a process that is immune to hangups
ps	POSIX.2	XPG4.2	Return the status of a process
renice	POSIX.2	XPG4.2	Change priorities of a running process
sleep	POSIX.2	XPG4.2	Suspend execution of a process for an interval of time
stop	POSIX.2	XPG4.2	Suspend a process or job
suspend	POSIX.2	XPG4.2	Send a SIGSTOP to the current shell
time	POSIX.2	XPG4.2	Display processor and elapsed times for a command
times	—	XPG4.2	Get process and child process times
wait	POSIX.2	XPG4.2	Wait for a child process to end
ulimit	—	XPG4.2	Set process limits

Writing Shell Scripts

autoload	—	—	Indicate function name not defined
dspmsg	—	—	Display selected messages from message catalogs
functions	—	—	Display or assign attributes to functions
getconf	POSIX.2	XPG4.2	Get configuration values

getopts	POSIX.2	XPG4.2	Parse utility options
integer	—	—	Mark each variable with an integer value
read	POSIX.2	XPG4.2	Read a line from standard input
type	—	XPG4.2	Tell how the shell interprets a name
typeset	—	—	Assign attributes and values to variables
xargs	POSIX.2	XPG4.2	Construct an argument list and run a command

Developing or Porting Application Programs

ar	POSIX.2	XPG4.2	Create or maintain library archives
awk	POSIX.2	XPG4.2	Process programs written in the **awk** language
c89	POSIX.2	XPG4.2	Compile C/MVS source code and create an executable file
c++	—	—	Compile C++/MVS and C/MVS and create an executable file
cc	—	XPG4.2	Compile common usage source code and create an executable file
dbx	—	—	Use the debugger
dspcat	—	—	Display all or part of a message catalog
gencat	—	XPG4.2	Create or edit message catalogs
lex	POSIX.2	XPG4.2	Generate a program for lexical tasks
make	POSIX.2	XPG4.2	Maintain program-generated and interdependent files
mkcatdefs	—	—	Preprocess a message source file
runcat	—	—	Pipe output from mkcatdefs to gencat
strip	POSIX.2	XPG4.2	Remove unnecessary information from an executable file
yacc	POSIX.2	XPG4.2	Use the **yacc** compiler

Communicating with the System or Other Users

mail	—	XPG4.2	Read and send mail messages
mailx	POSIX.2	XPG4.2	Send or receive electronic mail
mesg	POSIX.2	XPG4.2	Allow or refuse messages
talk	POSIX.2	XPG4.2	Talk to another user
write	POSIX.2	XPG4.2	Write to another user

Working with Archives

ar	POSIX.2	XPG4.2	Create or maintain library archives
cpio	—	XPG4.2	Copy in/out file archives
pax	POSIX.2	XPG4.2	Interchange portable archives
tar	—	XPG4.2	Manipulate the **tar** archive files to copy or back up a file

Working with UUCP

uucc	—	—	Compile UUCP configuration files
uucico	—	—	Process UUCP file transfer requests

uucp	—	XPG4.2	Copy files between remote UUCP systems
uucpd	—	—	Invoke **uucico** for TCP/IP connections from remote UUCP systems
uulog	—	XPG4.2	Display log information about UUCP events
uuname	—	XPG4.2	Display list of remote UUCP systems
uupick	—	XPG4.2	Manage files sent by uuto and uucp
uustat	—	XPG42	Display status of pending UUCP transfers
uuto	—	XPG42	Copy files to users on remote UUCP systems
uux	—	XPG42	Request command execution on remote UUCP systems
uuxqt	—	—	Carry out command requests from remote UUCP systems

Appendix B. tcsh Shell Command Summary

The following list presents the built-in tcsh shell commands, grouped by the task a user might want to perform, and their functions. Similar tasks are organized together.

General Use

alloc	—	—	Show the amount of dynamic memory acquired
builtins	—	—	Print the names of all built-in commands
bye	—	—	Terminate the login shell
echo	—	—	Write arguments to standard output
echotc	—	—	Exercise the terminal capabilities in args
exec	—	—	Run a command and open, close, or copy the file descriptors
glob	—	—	Write each word to standard output
hashstat	—	—	Print a statistic line on hash table effectiveness
login	—	—	Terminate a login shell
logout	—	—	Terminate a login shell
nice	—	—	Run a command at a different priority
notify	—	—	Notify user of job status changes
repeat	—	—	Execute command count times
source	—	—	Read and execute commands from name
time	—	—	Display processor and elapsed times for a command
where	—	—	Report all instances of command
which	—	—	Display next executed command

Controlling Your Environment

@ (at)	—	—	Print the value of tcsh shell variables, or assign a value
alias	—	—	Display or create a command alias
bindkey	—	—	List all bound keys, or change key bindings
complete	—	—	List completions
history	—	—	Display a command history list
hup	—	—	Run command so it exits on a hang-up signal
newgrp	—	—	Change to a new group
onintr	—	—	Control the action of the tcsh shell on interrupts
printenv	—	—	Display the values of environment variables
rehash	—	—	Recompute internal hash table
sched	—	—	Print scheduled event list
set	—	—	Set or unset command options and positional parameters
setenv	—	—	Set environment variable name to value
settc	—	—	Tell tcsh shell the terminal capability cap value
setty	—	—	Control tty mode changes
shift	—	—	Shift positional parameters
telltc	—	—	List terminal capability values
unalias	—	—	Remove alias definitions
uncomplete	—	—	Remove completions whose names match pattern
unhash	—	—	Disable use of internal hash table
unlimit	—	—	Remove resource limitations
unset	—	—	Unset values and attributes of variables and functions
unsetenv	—	—	Remove environment variables that match pattern

watchlog	—	—	Report on users who are logged in.

Managing Directories

cd	—	—	Change the working directory
chdir	—	—	Change the working directory
dirs	—	—	Print the directory stack
popd	—	—	Pop the directory stack
pushd	—	—	Make exchanges within directory stack

Computing and Managing Logic

break	—	—	Exit from a loop in a shell script
breaksw	—	—	Cause a break from a switch
continue	—	—	Skip to the next iteration of a loop in a shell script
default	—	—	Label default case in a switch statement
eval	—	—	Construct a command by concatenating arguments
exec	—	—	Run a command and open, close, or copy the file descriptors
exit	—	—	Return to the shell's parent process or to TSO/E
filetest	—	—	Apply a file inquiry operator to a file

Managing Files

ls-F	—	—	List files

Controlling Processes

bg	—	—	Move a job to the background
fg	—		Bring a job into the foreground
jobs	—	—	Return the status of jobs in the current session
kill	—	—	End a process or job, or send it a signal
limit	—	—	Limit consumption of processes
nohup	—	—	Start a process that is immune to hangups
stop	—	—	Suspend a process or job
suspend	—	—	Send a SIGSTOP to the current shell
time	—	—	Display processor and elapsed times for a command
wait	—		Wait for a child process to end

Appendix C. Advanced vi Topics

After you have mastered basic usage of **vi**, as described in “Using the vi Screen Editor” on page 255, you may want to explore some of the editor’s other capabilities.

Editing Options

vi has many options that change the way the editor behaves during an editing session. We will discuss a few that may be immediately useful. For a complete list of these options, see *OS/390 UNIX System Services Command Reference*.

You must be in Command Mode to set options. To set an option, begin by typing a colon (:). You will see the cursor move to the bottom of the screen. Then type the word `set`, a space, and the name of the option you want to set—we will talk about option names in a moment. You can correct typing mistakes can by backspacing. When you have typed everything correctly, press <Enter>.

One commonly used option is ″ignorecase″. If you type:

```
:set ignorecase
```

vi will not pay attention to the case of letters when searching. Many people prefer “caseless” searches over “case-sensitive” ones. If you want to go back to case-sensitive searches, type

```
:set noignorecase
```

Setting Tab Stops

By default, **vi** sets tab stops every 8 spaces. For example, if you begin a paragraph by typing a tab, the tab moves the cursor over 8 spaces. Many people feel 8 spaces are too many for a tab stop. You can set tab stops of 5 spaces with:

```
:set tabstop=5
```

Similar commands can set tabstops to any number of spaces.

Using Abbreviations

You can define an abbreviation for commonly used words or phrases. For example, if you type:

```
:ab www World Wide Web
```

and then press <Enter>, this sets the abbreviation. As soon as you type the abbreviation in text and move the cursor to the next space after `www`, the abbreviation is expanded into the associated phrase. The abbreviation function is case-sensitive. An abbreviation lasts the duration of a **vi** session. For information on how to set up a file with frequently used editing options, see “Setting Up an Editing Options Command File” on page 316.

If you want to get rid of an abbreviation that has been set, use the **:una** (unabbreviate) command. For example, type:

```
:una www
```

to get rid of the abbreviation.

Other Editing Options

For a complete list of the editing options, see *OS/390 UNIX System Services Command Reference*.

Setting Up an Editing Options Command File

A command file contains a number of commands that can be executed as if they were typed in a **vi** session. For example, you might use **vi** to create a file with the contents:

```
set wrapmargin=8
set tabstop=5
set shiftwidth=5
ab www World Wide Web
```

This sets all the options you want to use and all the abbreviations you need. The file can only contain instructions that normally start with a colon (:) in **vi**, but you omit the colons in the command file. During a **vi** session, you can execute all the instructions in the command file with the instruction:

```
:so cmdfile
```

where *cmdfile* is the name of your command file. **so** stands for source and it tells **vi** that the given file should be taken as the source of a number of commands.

You can execute the commands in a command file when you first start **vi**. Start **vi** with the command:

```
vi -c 'so cmdfile' filename
```

where *cmdfile* is the name of your command file and *filename* is the name of the file you want to edit. You might want to set up an alias for `vi -c 'so cmdfile'`; for example:

```
alias vic="vi -c 'so cmdfile'"
```

You can also set up a $HOME/.exrc file that contains all the commands you may want to run whenever you enter vi.

Editing Several Files

In a typical **vi** session, you may want to edit several files. When you have finished editing one file, you must first save your text in that file. Once you have saved your changes, you can start editing a different file by typing:

```
:edit newfilename
```

and then press <Enter>. This will clear out the text you have been editing and set things up so you can edit the new file. If the file already exists, its current contents will be read in.

Here's a trick to remember when you want to edit a number of files. If you start **vi** with a command line of the form

```
vi file1 file2 file3 ...
```

you can edit several files one after the other. After you have finished editing a file and saved it, you can move among files using the following commands:

Command	Action
:n	Edits the next file in the list of files.

:n!	Edits the next file in the list of files and discards the changes made to the current file.
:n *filenames*	Specifies a new list of files to be edited.

It may be particularly useful to use wild card characters on the **vi** command line, as in

```
vi *.c
```

This is expanded to a list of all the files under the current catalog that have the .c extension.

Combining Files

Occasionally, you may want to combine a number of files into a single document. For example, you may have a table of data stored in one file and wish to add the table to another file. You can read in the contents of a file after the line that holds the cursor. The **r** stands for Read; it reads the contents of a file to be added to the current file after the line indicated by the cursor.

The same sort of command may be used to combine the chapters of a document into a single file. For example,

```
:r chapter1
G
:r chapter2
G
:r chapter3
```

will read in chapters that are stored in separate files. Notice that we had to use **G** commands to go to the end of the file after each read operation, so that the next input file would be added to the end of the text.

Editing Program Source Code

Because **vi** originated on a UNIX system, the editor has a number of features primarily aimed at programming in the C language. However, these same features are applicable to many other languages.

Controlling Indention

The source code for a program differs from ordinary text in a number of ways. One of the most important of these is the way in which source code uses indention. Indentation shows the logical structure of the program: the way in which statements are grouped into blocks.

Issue the command:

```
:set autoindent
```

(Don't forget to press <Enter> after you have typed this.) The command turns on an option primarily supplied to control indention when entering source code. Each line is automatically indented the same distance as the previous one. As a programmer, you will find this saves you quite a bit of work getting the indention right, especially when you have several levels of indention.

When you are entering code with autoindent enabled, typing <EscChar-T> gives you another level of indention and typing <EscChar-D>takes one away. While you are in Insert Mode (*not* Command Mode):

- Type <EscChar-T> at the start of a line and it will be indented in another level.
- Type <EscChar-D> at the start of a line and the indention moves out one level of indent.

The amount of indention provided by <EscChar-T> is one tab character; the space depends on what **tabstop** is set at.

Try using the **autoindent** option when you are entering source code. It simplifies the job of getting indention correct. It can even sometimes help you avoid bugs; for example, in C source code, you usually need one closing } for every level of indention you go backwards.

The **<<** and **>>** commands are also helpful when indenting source code:

>> Shifts a line right 8 spaces (that is, adds 8 spaces of indention)

<< Shifts a line left 8 spaces (that is, removes 8 spaces of indention)

You can shift a number of lines by typing the number followed by **>>** or **<<**. For example, typing **5>>** will indent five lines, including the line the cursor is on.

The default shift is 8 spaces (right or left). You can change this default with this command:

```
:set shiftwidth=4
```

It is convenient to have a shiftwidth that is the same size as the width between tab stops.

Searching for Opening and Closing Brackets

The characters **(**, **[**, **{**, and **<** can all be called opening brackets. When the cursor is resting on one of these characters, pressing the **%** key moves the cursor from the opening bracket forward to the corresponding closing bracket **)**, **]**, **}**, and **>**, keeping in mind the usual rules for nesting brackets. For example, if you moved the cursor to the first **(** in:

```
if ( cos(a i ) > sin(b i +c i ) )
               {
                   printf("cos and sin equal!");
               }
```

and pressed **%**, you would see the cursor jump to the parenthesis at the end of the line. This is the closing parenthesis that matches the opening one.

Similarly, if the cursor is on one of the closing bracket characters, pressing **%** will move the cursor backwards to the corresponding opening bracket character.

Not only does this search character help you move forward and backward through a program in long jumps, it lets you check the nesting of parentheses in source code. For example, if you put the cursor on the first **{** at the beginning of a C function, pressing **%** should move you to the **}** that (you think) ends the function. If it doesn't, something has gone wrong somewhere.

Making Substitutions

If the name of a data object or function has to be changed in a program (for whatever reason), it becomes necessary to change every occurrence of that name. This would be a tedious process using the **vi** features we have discussed up to this

point, because you would have to search through each source file for the name and then type in the new name wherever the old one was found. To avoid much of this work, **vi** offers the “substitute” command.

The usual form of the substitute command is

```
:s/pattern/replacement/
```

where *pattern* is any of the patterns used in searches and *replacement* is any string.

As soon as you type the colon (:), you see the cursor move to the bottom of the screen. Then type the rest of the command and press <Enter>. The command puts the given *replacement* string in the place of the first string that matches the given *pattern*.

What happens if a line has more than one string that matches the pattern? The **s** command replaces only the first occurrence of a given string on a line. The position of the cursor in the line does not matter.

If you want to change every occurrence of a string on a line, type a **g** (for “global”) after the last slash.

Specifying a Range of Lines to Change

In addition to applying **s** to a single line, you can apply it to a range of lines. For example, let’s examine the command:

```
:1,200s/^/!/
```

What happens? The **1,200** in front of the **s** indicates that the command should be applied to the lines from 1 through 200 (everything up to the 200th line in the file). The **s** command itself says to replace the beginning of the line (^) with an exclamation point. So an exclamation point would be put at the beginning of every line up to number 200. To get rid of the exclamation points, you would type:

```
:1,200s/^!//
```

which says change every ! at the beginning of a line into “nothing”.

Determining Line Numbers

In the above instructions, we made use of line numbers to refer to lines. How do you know what number a line has? If you just want to know the number of one line, move the cursor to that line and type

```
:.=
```

For another approach, type:

```
:set number
```

and press <Enter>. As you can now see, this displays the number of every line in the file. If you want to turn off the display of line numbers, type:

```
:set nonumber
```

A number of special symbols can be used when specifying a range of lines. The . (period) stands for the line where the cursor is currently positioned. For example, move the cursor to this line and type:

```
:1,.s/$/???/
```

This adds ??? to the end of every line from the start of the file to the line containing the cursor.

When you issue a substitute command with a range, it is all right if some of the lines in the range do not contain the pattern you are replacing. When specifying a range of lines, **$** stands for the last line in the file. For example, the command:

```
:1,$s/the/THE/g
```

changes every `the` in the file to uppercase (including words like `there` where `the` is part of another word).

Checking as You Substitute

What would you do now if you want to change the variable *i* into a *k*? You can't just use an instruction like

```
:254,267s/i/k/g
```

because that will change the letter *i* into *k* even in other words like **int** and **list**.

The solution to this is to add a **c** (for check) after the **s** command. For example,

```
:s/pattern/replacement/gc
```

When you do this, **vi** checks with you before making every substitution. Before each possible change, **vi** prints the line at the bottom of your screen and puts a ^ under the string that might be changed. If you want the change to happen, press the <Y> key followed by <Enter>. If you do not want the change to happen, press the <N> key followed by <Enter>.

Appendix D. Using awk

awk is a programming language that lets you work with information stored in files. With **awk** programs, you can:

- Display all the information in a file, or selected pieces of information
- Perform calculations with numeric information from a file
- Prepare reports based on information from a file
- Analyze text for spelling, frequency of words or letters, and so on

You can combine these operations to perform quite complicated tasks.

awk allows most of the logical constructs of modern computing languages: **if–else** statements, **while** and **for** loops, function calls, and so on.

This appendix introduces some of the principles and concepts of **awk**. The OS/390 version of awk is based on the POSIX definition of **awk** and also supports the functionality of **nawk**, the new **awk**. Experienced programmers may prefer to turn directly to **awk** in *OS/390 UNIX System Services Command Reference*. For an excellent reference for **awk**, see *The AWK Programming Language* by Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan (Addison-Wesley, 1988). Aho, Weinberger, and Kernighan are the people who created **awk** at AT&T Laboratories, and the name *awk* comes from their last names.

Data Files

awk programs work with *data*. Programs can obtain data typed in from the workstation or from the output of other commands (for example, through *pipes*), but usually data is obtained from *data files*.

awk's data files are always text files (not binary files). The files contain readable text—for example, words, numbers, punctuation characters, and so on.

As an example, consider a data file named **hobbies**, which contains information on the hobbies of a group of people. Each line in this file gives a person's name, one of that person's hobbies, how many hours a week he or she spends on the hobby, and how much money the hobby costs per year. One hobby per person appears on each separate line. The file might look like this:
This file is included with the OS/390 UNIX shell as **/samples/hobbies**.

```
Jim     reading          15     100.00
Jim     bridge            4      10.00
Jim     role playing      5      70.00
Linda   bridge           12      30.00
Linda   cartooning        5      75.00
Katie   jogging          14     120.00
Katie   reading          10      60.00
John    role playing      8     100.00
John    jogging           8      30.00
Andrew  wind surfing     20    1000.00
Lori    jogging           5      30.00
Lori    weight lifting   12     200.00
Lori    bridge            2       0.00
```

Figure 34. The hobbies File

Records

An **awk** data file is a collection of *records*. A record contains a number of pieces of information about a single item; these pieces are called *fields*.

Records are separated by a *record separator character,* which, for **awk**, is usually the *newline* character. A newline character shows where one line of text ends and another begins; by using the newline as a record separator, each line of the file becomes a separate record. This is convenient and easy to understand; newline is used as a record separator in all of the examples.

In the **hobbies** file, each line is a separate record, giving a set of information about one person's hobby.

Fields

A record consists of a number of *fields*. A field is a single piece of information. For example, the hobby record:

```
Jim     reading          15     100.00
```

contains four fields:

```
Jim
reading
15
100.00
```

Fields should be provided in the same order in each record. That way **awk** and other programs can easily access a particular piece of information in any record.

The fields of a record are separated by one or more *field separator characters*. The **hobbies** file uses strings of blank characters (spaces) to separate fields. By default, **awk** uses blanks or horizontal tab characters to separate fields. You can change the default.

The Shape of a Program

An **awk** program looks like this:

```
pattern {actions}
pattern {actions}
pattern {actions}
  ...
```

Each line is a separate instruction. **awk** looks through the data files record by record and executes the instructions, in the given order, on each record.

Simple Patterns

A instruction of the form:

```
pattern {actions}
```

indicates that **awk** is to perform the given set of actions on every record that meets a certain set of conditions. The conditions are given by the *pattern* part of the instruction.

The *pattern* of an instruction often looks for records that have a particular value in some field. The notation `$1` stands for the first field of a record, `$2` stands for the second field, and so on. For example, here's a simple **awk** instruction:

```
$2 == "jogging" { print }
```

The notation == stands for “is equal to”. Therefore, the instruction means: *If the second field in a record is* jogging*, print the entire record.*

This instruction is a complete **awk** program. If you ran this program on the **hobbies** file, **awk** would look through the file record by record (line by line). Whenever a line had `jogging` as its second field, **awk** would print the complete record. The printout from the program would be:

```
Katie   jogging          14     120.00
John    jogging           8      30.00
Lori    jogging           5      30.00
```

Let’s take another example. Ask yourself what the following **awk** program does.

```
$1 == "John" { print }
```

As you probably guessed, it prints every record that has `John` as its first field. The printout from the program would be:

```
John    role playing      8     100.00
John    jogging           8      30.00
```

You could perform the same sort of search on any text database. The only difference is that databases tend to contain a great deal more data than this example.

If an **awk** instruction does not contain an action, **print** is assumed. The preceding examples both use the **print** action; however, this action does not need to be written explicitly. Therefore, you could write the programs as:

```
$2 == "jogging"
```

and:

```
$1 == "John"
```

and they would have exactly the same effect.

On the other hand, you can specify an action and leave out the pattern part of an instruction. In this case, **awk** applies the action part of the instruction to every record in the file. For example:

```
{ print }
```

is a complete **awk** program that displays every record in the data file.

Using Blanks and Horizontal Tabs

You can put any number of extra blanks or horizontal tabs into **awk** patterns and actions.

Note: If you are using the ISPF/PDF editor to write an **awk** program and want to use horizontal tabs, see “Typing Tabs in ISPF” on page 247.

For example, you can enter:

```
{ print $1 , $2 , $3 }
```

Applying More Than One Instruction

When an **awk** program contains several instructions, **awk** applies every appropriate instruction to the first record, then every appropriate instruction to the second record, and so on. Instructions are applied in order. For example, consider the following **awk** program, which has two instructions:

```
$1 == "Linda"
$2 == "bridge" { print $1 }
```

The output of this program is:

```
Jim
Linda   bridge           12      30.00
Linda
Linda   cartooning        5      75.00
Lori
```

awk looks through the file record by record. The first record to satisfy one of the patterns is:

```
Jim     bridge            4      10.00
```

so **awk** prints the first field of the record (as dictated by the second instruction). The next record of interest is:

```
Linda   bridge           12      30.00
```

This satisfies the first instruction's pattern, so the whole record is printed. It also satisfies the second instruction's pattern, so the first field is printed. **awk** continues through the file, record by record, executing the appropriate actions when a record satisfies the pattern.

Assigning Values to Variables

Suppose you want to find out how many people have jogging as a hobby. To do this, you have to look through the **hobbies** file, record by record, and keep a count of the number of records that have `jogging` in their second field. This means that you have to *remember* the count from one record to the next.

awk programs *remember* information by using *variables*. A variable is a storage place for information. Every variable has a name and a value. An **awk** action of the form:

```
name = value
```

assigns the specified *value* to the variable that has the given *name*. For example:

```
count = 0
```

assigns the value `0` to the variable *count*.

You can use variables in expressions. For example, the value of the expression:

```
count + 1
```

is the current value of *count*, plus `1`.

String Values

A *string value* is just a sequence of characters like `"abc"`. A string value is always enclosed in quotes. Any sort of characters are allowed (even digits, as in `"abc123"`). Strings can contain any number of characters. A string with zero characters is called the *null string* and is written `""`.

When **awk** compares strings, it makes comparisons in accordance with the collating order set by the locale defined on the system. This is a little like alphabetic order; for example, the program:

```
$1 >= "Katie"
```

prints the `Katie`, `Linda`, and `Lori` lines, which is what you would expect from alphabetic order. However, collating orders differ. ASCII collating order, for example, differs from alphabetic order in a number of respects; for example, lowercase letters are “greater” than uppercase ones, so that *a* is greater than *Z*.

Numeric Values

A *numeric value* consists of digits with an optional sign and decimal point. A numeric values is not enclosed in quotes. For example:

```
10    0.34    -78    +2.56    -.92
```

are all valid in **awk**. **awk** does not let you put commas inside numbers. For example, you must write 1000 instead of 1,000.

Note: **awk** lets you use exponential or scientific notation. Exponents are given as `e` or `E`, followed by an optionally signed exponent. Thus:

```
1E3    1.0e3    10E2    1000
```

are all equivalent.

When **awk** compares numbers (with such operators as > or <), it makes comparisons in accordance with the usual rules of arithmetic.

Using the print Action for Output

So far, **print** has been the only action discussed. As you have seen, **print** can display an entire record. It can also display selected fields of the record, as in:

```
$2 == "bridge" { print $1 }
```

This displays the first field of every record with a second field that is `bridge`. The output is:

```
Jim
Linda
Lori
```

print can display more than a single field. If you give **print** a list of fields separated by commas, as in:

```
$1 == "Jim" { print $2,$3,$4 }
```

print displays the given fields separated by single blanks, as in:

```
reading 15 100.00
bridge 4 10.00
role playing 5 70.00
```

The **print** action can display strings and numbers along with fields. For example:

```
$1 == "John" { print "$",$4 }
```

prints:

```
$ 100.00
$ 30.00
```

In this instruction, the **print** action prints a string containing a `$`, followed by a blank, followed by the value of the fourth field in each selected record.

As an exercise, predict the output of the following:

```
(a)  $1 == "Lori" { print $1,"spends $", $4,"on",$2 }
(b)  $2 == "jogging" { print $1,"jogs",$3,"hours a week" }
(c)  $4 > 100.00 { print $1, "has an expensive hobby" }
```

You can check your predictions by running these programs against the **hobbies** file.

Running awk Programs

There are two ways to run **awk** programs: from a command line and from a program file.

The awk Command Line

The simplest **awk** command line is:

```
awk 'program' datafile
```

The **awk** program is enclosed in single-quote or apostrophe (') characters. The *datafile* argument gives the name of the data file. For example:

```
awk '$1 == "Linda"' hobbies
```

executes the program:

```
$1 == "Linda"
```

on the data file **hobbies**.

If you are using the OS/390 shell, you can type in a multiline program within single quotes, as in:

```
awk '
   $1 == "Linda"
   $2 == "bridge" { print $1 }
   ' hobbies
```

awk assumes that blanks or horizontal tabs separate fields in a record. If the data file uses different field separator characters, you must indicate this on the command line. You can do this with an option of the form:

```
–Fstring
```

where *string* lists the characters used to separate fields. For example:

```
awk –F":" '{ print $3 }' file.dat
```

indicates that the given data file uses colon (**:**) characters to separate record fields. The **–F** option must come before the quoted program instructions.

awk also allows you to define the value of variables on the command line by using the **–v** option. See *OS/390 UNIX System Services Command Reference* for details.

Program Files

A program file is a text file that contains an **awk** program. You can create program files with any text editor (such as **ed**). For example, you might create a file named **lbprog.awk** that contains the lines:

```
$1 == "Linda"
$2 == "bridge" { print $1 }
```

To execute a program on a particular data file, use the command:

```
awk –f progfile
datafile
```

where *progfile* is the name of the file that contains the **awk** program and *datafile* is the name of the data file. For example:

```
awk –f lbprog.awk hobbies
```

runs the program in **lbprog.awk** on the data in **hobbies**.

If the data file does not use the default separator characters, you must specify a **–F** option after the *progfile* name, as in:

```
awk –f prog.awk –F":" file.dat
```

To gain some experience using **awk**, you can test the examples on the **hobbies** file. Run some from the command line and some from program files.

Sources of Data

If you do not specify a data file on the command line, **awk** begins to read data from standard input. For example, if you enter the command:

```
awk '{ print $1 }'
```

awk prints the first word of every line you type. When you type in data from the workstation, press <Enter> at the end of each line. To stop passing data to **awk**, type <EscChar-D> and press <Enter>.

A command line may also specify several data files, as in:

```
awk -f progfile data1 data2 data3 ...
```

When **awk** has finished reading through the first data file **data1**, it goes on to **data2**, and so on.

Operators

awk recognizes these types of operators:

- Comparison operators
- Arithmetic operators
- Compound assignments
- Increment and decrement operators
- Matching operators
- Multiple-condition operators

Comparison Operators

The == notation is an example of a *comparison*. **awk** recognizes several types of comparisons:

Operator	Meaning
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Arithmetic Operators

The following **awk** program uses simple arithmetic:

```
$3 > 10 { print $1, $2, $3–10 }
```

In the **print** statement:

```
$3–10
```

has the value of the third field in the record, minus 10. This is the value that **print** prints. If you apply this program to the **hobbies** file, the output is:

```
Jim reading 5
Linda bridge 2
Katie jogging 4
Andrew wind surfing 10
Lori weight lifting 2
```

You could describe how the program works like this: If someone spends more than 10 hours on a hobby, the program prints the person's name, the name of the hobby, and how many *extra* hours the person spends on the hobby (that is, the number of hours more than 10).

An expression such as:

```
$3–10
```

is called an *arithmetic expression*. It performs an arithmetic operation and comes up with a result, which is called the *value* of the expression.

awk recognizes the following arithmetic operations:

Operation	Operator	Example
Addition	*A* + *B*	2+3 is 5
Subtraction	*A* – *B*	7–3 is 4
Multiplication	*A* * *B*	2*4 is 8
Division	A / B	6/3 is 2
Negation	– *A*	– 9 is –9
Remainder	*A* % *B*	7%3 is 1
Exponentiation	*A* ^ *B*	3^2 is 9

The remainder operation is also known as the *modulus,* or *integer remainder* operation. The value of this expression is the integer remainder you get when you divide *A* by *B*. For example:

```
7 % 3
```

has a value of 1, because dividing 7 by 3 gives you 2 with a remainder of 1.

The value for the *exponentiation* operation:

```
A ^ B
```

is the value of *A* raised to the exponent *B*. For example:

```
3 ^ 2
```

has the value 9 (that is, 3^2).

Operation Ordering

Expressions can contain several operations, as in:

```
A+B*C
```

As is customary in mathematics, all multiplications and divisions and remainder operations are performed *before* additions and subtractions. When handling the foregoing expression, **awk** performs `B*C` first and then adds `A`. The value of:

```
2+3*4
```

is therefore 14 (3*4 first, then add 2). If you want a particular operation done first, enclose it in parentheses, as in:

```
(A+B)*C
```

When evaluating this expression, **awk** performs the addition before the multiplication. Therefore:

```
(2+3)*4
```

is 20 (2+3 first, then multiply by 4). As an example of this, consider the program:

```
{ print $4/($3*52) }
```

`$4` is the amount of money a person spent on a hobby in the last year. `$3` is the average number of hours a week the person spent on that hobby, so `$3*52` is the number of hours in 52 weeks (that is, 1 year). `$4/($3*52)` is therefore the amount of money that the person spent on the hobby *per hour*.

An order-of-operations table for **awk** can be found in **awk** in *OS/390 UNIX System Services Command Reference*.

Compound Assignments

The following are the compound assignment operations of **awk** and their equivalents:

Compound Operation	Equivalent
A += B	A = A + B
A –= B	A = A – B
A *= B	A = A * B
A /= B	A = A / B
A %= B	A = A % B
A ^= B	A = A ^ B

Increment and Decrement Operators

You can advance the value held in a variable, with:

```
count = count + 1
```

This is such a common operation that **awk** has a special operator for incrementing variables by 1.

++ The ++ operator increments the current value of the variable by 1. For example:

```
count++
```

adds 1 to the current value of *count*.

– – The – – decrements (subtracts 1 from) the current value of a variable. For example, to subtract 1 from *count*, write:

```
count--
```

Matching Operators

If the pattern in an instruction is just a regular expression, **awk** looks for a matching string anywhere in a record. Sometimes, however, you want to look for a matching string only in a particular field of a record. In this case, you can use a *matching* expression.

There are two types of matching expressions:

string ~ /*regular-expression*/
: Is true if *string* matches the given regular expression. (The ~ character is called “tilde.”)

string !~ /*regular-expression*/
: Is true if *string* does not match the given regular expression.

Multiple-Condition Operators

Operator
: **Meaning**

&&
: The double ampersand operator means **AND**. For example:

```
$3 > 10 && $4 > 100.00 { print $1, $2 }
```

prints the first and second fields of any record where `$3` is greater than 10 *and* `$4` is greater than 100.00.

||
: The double “or-bar” operator means **OR**. For example:

```
$1 == "Linda" || $1 == "Lori"
```

prints any record with a first field that is *either* `Linda` *or* `Lori`.

Regular Expressions

A regular expression is a way of telling **awk** to select records that contain certain strings of characters. For example, the instruction:

```
/ri/ { print }
```

tells **awk** to print all records that contain the string `ri`. Regular expressions are always enclosed in *slashes* as shown in the instruction just discussed. For a discussion of regular expressions beyond their usage in **awk**, see the appendix on regular expressions in *OS/390 UNIX System Services Command Reference*.

The following characters have special meanings when you use them in regular expressions.

^
: Stands for the beginning of a field. For example:

```
$2 ~ /^b/ { print }
```

Prints any record whose second field begins with `b`.

$
: Stands for the end of a field. For example:

```
$2 ~ /g$/ { print }
```

prints any record with a second field that ends with `g`.

.
: Matches any single character (except the newline). For example:

```
$2 ~ /i.g/ { print }
```

selects the records with fields containing `ing`, and also selects the records containing `bridge` (`idg`).

| Means *or*. For example:

```
/Linda|Lori/
```

is a regular expression that matches either of the strings `Linda` or `Lori`.

***** Indicates zero or more repetitions of a character. For example:

```
/ab*c/
```

matches `abc`, `abbc`, `abbbc`, and so on. It also matches `ac` (zero repetitions of `b`). Since **.** matches any character except the newline, **.*** matches an arbitrary string of zero or more characters. For example:

```
$2 ~ /^r.*g$/ { print }
```

prints any record with a second field that begins with `r`, ends in `g`, and has any set of characters between (for example, `reading` and `role playing`).

+ Is similar to `*`, but stands for *one* or more repetitions of a character. For example:

```
/ab+c/
```

matches `abc`, `abbc`, and so on, but does not match `ac`.

\{*m,n***\}**
Indicates *m* to *n* repetitions of a character (where *m* and *n* are both integers). For example:

```
/ab\{2,4\}c/
```

would match `abbc`, `abbbc`, and `abbbbc`, and nothing else.

? Is similar to `*`, but stands for zero or one repetitions of a string. For example:

```
/ab?c/
```

matches `ac` and `abc`, but not `abbc`, and so on.

[X] Matches any one of the set of characters *X* given inside the square brackets. For example:

```
$1 ~ /^[LJ]/ { print }
```

prints any record whose first field begins with either `L` or `J`. As a special case: `[:lower:]` inside the square brackets stands for any lowercase letter, `[:upper:]` inside the square brackets stands for any uppercase letter, `[:alpha:]` inside the square brackets stands for any letter, and `[:digit:]` inside the square brackets stands for any digit.

Thus:

```
/[[:digit:][:alpha:]]/
```

matches a digit or letter.

[^X] Matches any one character that is not in the set *X*. For example:

```
$1 ~ /^[^LJ]/ { print }
```

prints any record with a first field that does not begin with `L` or `J`.

```
$1 ~ /^[^[:digit:]]/ { print }
```

prints any record with a first field that does not begin with a digit.

(X) Matches anything that the regular expression *X* does. You can use parentheses to control how other special characters behave. For example, `*` normally applies to the single character immediately preceding it. This means that:

```
/abc*d/
```

matches `abd`, `abcd`, `abccd`, and so on. However:

```
/a(bc)*d/
```

matches `ad`, `abcd`, `abcbcd`, `abcbcbcd`, and so on.

The characters with special meanings are:

```
^   $   .   *   +   ?   [   ]   (   )   |
```

These are known as *metacharacters*.

When a metacharacter appears in a regular expression, it usually has its special meaning. If you want to use one of these characters literally (without its special meaning), put a backslash in front of the character. For example:

```
/\$1/ { print }
```

prints all records that contain a dollar sign `$` followed by a `1`. If you simply entered:

```
/$1/ { print }
```

awk would search for records where the end of the record was followed by a `1` — which is impossible.

Because the backslash has this special meaning, `\` is also considered a metacharacter. If you want to create a regular expression that matches a backslash, you must therefore use two backslashes `\\`.

Pattern Ranges

A instruction of the form:

```
pattern1, pattern2 { action }
```

performs the given *action* on every line, starting at an occurrence of *pattern1* and ending at the next occurrence of *pattern2* (inclusive). For example, the instruction

```
/Jim/, /Linda/ { print $2 }
```

prints the second field of all lines between an occurrence of `Jim` and an occurrence of `Linda`. Using the **hobbies** file as our data file, the output is:

```
reading
bridge
role playing
bridge
```

When **awk** finds a record matching *pattern2,* it begins to look for a line matching *pattern1* again. Thus, with this instruction:

```
/reading/, /role/
```

the output is

```
Jim     reading          15     100.00
Jim     bridge            4      10.00
Jim     role playing      5      70.00
Katie   reading          10      60.00
John    role playing      8     100.00
```

awk prints the first range of records from `reading` to `role` and then starts looking for `reading` again.

awk starts performing the instruction's action as soon as there is a record that matches *pattern1*. **awk** does not check to make sure that there is a line matching *pattern2* in the rest of the file. This means that:

```
/Lori/, /Jim/ { print $2 }
```

begins printing at the first record that contains `Lori`, and keeps going until it reaches the end of the file. No `Jim` is found.

Using Special Patterns

`BEGIN` and `END` are two special patterns.

BEGIN When an instruction has `BEGIN` as its pattern, **awk** performs the associated action *before* looking at any of the records in the data file.

END When an instruction has `END` as its pattern, **awk** performs the associated action *after* looking at all records in the data files specified on the command line.

Consider the action:

```
count = count + 1
```

awk first finds the value of:

```
count + 1
```

and then assigns this value to *count*. Thus this action increases the value of *count* by 1. In a program, you can use this sort of action to count how many people have jogging as a hobby:

```
BEGIN { count = 0 }
$2 == "jogging" { count = count + 1 }
END { printf "%d people like jogging.\n", count }
```

Let's look at this program line by line.

```
BEGIN { count = 0 }
```

In this example, **awk** begins by assigning the value `0` to *count*:

```
$2 == "jogging" { count = count + 1 }
```

adds 1 to *count* every time **awk** finds a record with `jogging` in the second field.

```
END { printf "%d people like jogging.\n", count }
```

When **awk** has looked at all the records, the **printf** action prints the count of people who jog. The output from the program is:

```
3 people like jogging.
```

Notice how the value of *count* was printed in place of the **%d** placeholder. For more information about using a placeholder, see “Placeholders” on page 342.

Built-in Variables

awk has a number of *built-in variables* that you can use in your programs. You do not have to assign values to these variables; **awk** automatically assigns the values for you.

Built-in Numeric Variables

The following list describes some of the important numeric built-in variables:

NR Contains the number of records that have been read so far. When **awk** is looking at the first record, `NR` has the value 1; when **awk** is looking at the second record, `NR` has the value 2; and so on. In a `BEGIN` instruction, `NR` has the value `0`. In an `END` instruction, `NR` contains the total number of records that were read. This instruction:

```
END  { print NR }
```

prints the total number of data records read by the **awk** program.

FNR Is like `NR`, but it counts the number of records that have been read so far *from the current file*. When you give several data files on the **awk** command line, **awk** sets `FNR` back to 1 when it begins reading each new file. Thus, a command such as:

```
{ printf "%d:%s\n",FNR,$0 }
```

prints the line number in the current file, followed by a colon, followed by the contents of the current line.

NF Gives the number of fields in the current record. For the **hobbies** file, `NF` is 4 for each line, because there are four fields in each record. In an arbitrary text file, `NF` gives the number of words on the current line in the file; by default, **awk** assumes that blanks separate the fields of a record, so it considers each word on a line to be a separate field. Therefore, the program:

```
{ count = count + NF }
END  { print count }
```

prints the total number of words in the file.

Using these built-in variables, you can create more ambitious **awk** commands.

```
awk 'NF == 1 {print}' file
```

prints those records with precisely one field in them. There is no **–F** option specified for this command, so **awk** assumes that blanks or tab characters separate the fields. The foregoing command therefore prints all lines that contain only one word (that is, one field).

```
awk '{print FNR ": " $0}' file
```

`$0` stands for the entire record. The foregoing command displays the contents of **file**, putting a line number and a colon before each line.

```
awk '/abc/ {print FILENAME ": " $0}' *.bas
```

examines all files that have the **.bas** extension in the working directory. It prints every line that contains the string `abc` and also displays the filename, so you know which file contains which lines.

Built-in String Variables

awk also provides a number of built-in string variables:

FILENAME
Contains the name of the current input file. For example, when running programs against the **hobbies** file, the value of `FILENAME` would be **hobbies** (if that is the file you are using). If the input is coming from the **awk** standard input, the value is `-`.

FS Is the *field separator* string, giving the character that is used to separate fields in the current file. The default value for `FS is ""` (a single blank), which as a special case matches both blank and tab. However, if the command line contains an **–F** option specifying a different field separator, `FS` is a string containing the given separator character. As well, a program may also assign values to `FS` to indicate new field separator characters. For example, you could create a data file with a first line that provides the character used to separate fields in the records in the rest of the file. An **awk** program could then contain the instruction:

```
FNR == 1 { FS = $0 }
```

This says that the field separator string `FS` should be assigned the contents of the first record in the current data file. The character in this line is then taken to be the field separator for the rest of the file (unless `FS` changes value again). Any `FS` value of more than one character is used as a regular expression. For details, see the “Input” section of the **awk** command description in *OS/390 UNIX System Services Command Reference*.

RS Is the *input record separator*. Just as `FS` indicates the character that separates fields within records, `RS` indicates the character that separates one record from another. By default, `RS` contains a newline character, which means that input records are separated by newlines. However, you can assign a different character to `RS`; for example, with:

```
RS = ";"
```

input records are separated by semicolons. This lets you have several records on a single line, or a single record that extends over several lines. Records are separated by a semicolon, not a <newline> character. As an important special case:

```
RS = ""
```

separates records by empty lines.

OFS Gives the *output field separator string*. When you use the **print** action to print several values, as in:

```
{ print A, B, C }
```

awk prints the output field separator string between each of the values. By default, `OFS` contains a single blank character, which is why output values are separated by a single blank. However, if you make the assignment:

```
OFS = " : "
```

the output values are separated by the given string. You can also use `OFS` to reconstruct the `$0` field during field assignment.

ORS Gives the *output record separator*. When you use the **print** action to print records, **awk** prints the output record separator at the end of each record. By default, `ORS` is the newline character, which is why **print** prints a new output line each time it is called. However, you can use a different separator string by assigning the string to `ORS`.

OFMT Is the *default output format* for numbers when they are displayed by **print**. This is a format string like the one used by **printf**. By default, it is `%.6g`, indicating that numbers are to be displayed with a maximum of six digits after the decimal point. By changing OFMT, you can obtain more or less displayed precision.

CONVFMT
Is the *default format* which **awk** uses when converting numbers into strings internally. This differs from the **OFMT** variable, which is used only when displaying numbers. The internal conversion of a number to a string occurs when you perform concatenation, indexing, and some comparison operations. **awk** converts floating-point numbers (that is, numbers that are not integers) to strings as if you had specified the operation:

```
sprintf(CONVFMT, number ...)
```

By default, the value of **CONVFMT** is `%.6g`.

Note: **CONVFMT** is a POSIX extension not found in traditional implementations of **awk**.

Statements and Loops

awk supports the following types of statements and loops:
- **if** statement
- **while** loop
- **for** loop
- **next** statement
- **exit** statement

The if Statement

An **if** statement is an action of the form:

```
if (expression) statement1 else
statement2
```

Typically, the *expression* in the **if** statement has a true-or-false value. If the value is true, *statement1* is performed; otherwise, *statement2* is performed. The **else** *statement2* part is optional.

The while Loop

A **while** loop repeats one or more other instructions as long as a given condition holds true. The format of the loop is:

```
while (expression) statement
```

where the statement can be a single statement or a compound statement.

The for Loop

The statement:

```
for
(expression1;expression2;expression3)
statement
```

is equivalent to the following instruction sequence:

```
expression1
while (expression2) {
    statement
    expression3
}
```

The next Statement

The **next** instruction skips immediately to the next record in the data file.

The exit Statement

The **exit** statement makes an **awk** program behave as if it had just reached the end of data input. No further input is read. If there is an `END` action, **awk** executes it before the program ends. As with **next**, **exit** is often used when input data is found to be incorrect.

If **exit** appears inside the `END` action, the program ends immediately.

Functions

awk supports:

- Arithmetic functions
- String manipulation functions
- User-defined functions
- Passing an array to a function
- The **getline** function

Arithmetic Functions

awk recognizes the most common mathematical functions, as shown in the following table.

Function	Result
sqrt(*x*)	Square root of *x*
sin(*x*)	Sine of *x,* where *x* is in radians
cos(*x*)	Cosine of *x,* where *x* is in radians
atan2(*y,x*)	Arctangent of *y/x* in range $-\pi$ to π
log(*x*)	Natural logarithm of *x*
exp(*x*)	The constant *e* to the power *x*
int(*x*)	Integer part of *x*
rand()	Random number between 0 and 1
srand(*x*)	Sets *x* as seed for **rand()**

Several of these functions may require more explanation.

The **int** function takes a floating-point number as an argument and returns an integer. The integer is just the floating-point number, without its fractional part.

Every call to **rand** returns a new random number between 0 and 1. In this way, you can get a sequence of random numbers. You can use **srand** to set the starting point, or “seed” for a random number sequence. If you set the seed to a particular

value, you always get the same sequence of numbers from **rand**. This is useful if you want a program to use **rand** but obtain uniform results every time the program runs.

String Manipulation Functions

awk has a number of functions that perform string operations:

length Returns an integer that is the length of the current record (that is, the number of characters in the record, without the newline on the end). For example, the following program calculates the total number of characters in a file (except for newline characters):

```
    { sum = sum + length }
END { print sum }
```

length(*s*)
Returns an integer that is the length of the string *s*. For example, the following program prints the length of the first field in each record of the file:

```
{ print length($1) }
```

The function call **length**($0) is equivalent to just **length**.

gsub(*regexp,replacement*)
Puts the replacement string *replacement* in place of every string matching the regular expression *regexp* in the current record. For example, the program:

```
{
    gsub(/John/,"Jonathan")
    print
}
```

checks every record in the data file for the regular expression John, replaces matching strings with Jonathan, and prints the resulting record. As a result, the program's output is exactly like its input, except that every occurrence of John is changed to Jonathan. This form of the **gsub** function returns an integer telling how many substitutions were made in the current record. This is 0 if the record has no strings that match *regexp*.

sub(*regexp,replacement*)
Is similar to **gsub** except that it replaces only the *first* occurrence of a string matching *regexp* in the current record.

gsub(*regexp,replacement,string_var*)
Puts the replacement string *replacement* in place of every string matching the regular expression *regexp* in the string *string_var*. For example, the program:

```
{
    gsub(/John/,"Jonathan",$1)
    print
}
```

is similar to the previous program, but the replacement is made only in the first field of each record. This form of the **gsub** function returns an integer telling how many substitutions were made in *string_var*.

sub(*regexp,replacement,string_var*)
Is similar to the previous version of gsub except that it only replaces the *first* occurrence of a string matching *regexp* in the string *string_var*.

Note: You must use four backslashes to embed one literal backslash in a **gsub()** or **sub()** substitution string. For example,

```
gsub(/backslash/,"\\\\")
```

replaces all occurrences of the word `backslash` with the single character \.

index(*string,substring***)**

Searches the given *string* for the appearance of the given *substring*. If it cannot find *substring*, **index** returns 0; otherwise, **index** returns the number (origin 1) of the character in *string* where *substring* begins. For example:

```
index("abcd","cd")
```

returns the integer 3 because `cd` is found beginning at the third character of `abcd`.

match(*string,regexp***)**

Determines if *string* contains a substring that matches the regular expression (pattern) *regexp*. If so, the function returns an index giving the position of the matching substring within *string*; if not, **match** returns 0. **match** also sets a variable named **RSTART** to the index where the matching string starts, and a variable named **RLENGTH** to the length of the matching string.

substr(*string,pos***)**

Returns the last part of *string* beginning at a particular character position. The argument *pos* is an integer, giving the number of a character. Numbering begins at 1. For example, the value of:

```
substr("abcd",3)
```

is the string `cd`.

substr(*string,pos,length***)**

Returns the part of *string* that begins at the character position given by *pos* and has the length given by *length*. For example, the value of:

```
substr("abcdefg",3,2)
```

is `cd` (a string of length 2 beginning at position 3).

sprintf(*format,value1,value2,...***)**

Is based on the **printf** action. The value of **sprintf** is the string that would be printed out by the action

```
printf(format,value1,value2,...)
```

For example:

```
str = sprintf("%d %d!!!\n",2,3)
```

assigns the string `"2 3!!!\n"` to the string variable `str`.

tolower(*string***)**

Returns the value of *string*, but with all the letters in lowercase. (This function is an extension to standard **awk**.)

toupper(*string***)**

Returns the value of *string*, but with all the letters in uppercase. (This function is an extension to standard **awk**.)

ord(*string***)**

Converts the first character of *string* into a number. This number gives the

decimal value of the character in the character set used on the system. (This function is an extension to standard **awk**.)

User-Defined Functions

In an **awk** program, a function definition looks like this:

```
function name(argument-list) {
    statements
}
```

The *argument-list* is a list of one or more names (separated by commas) that represent argument values passed to the function. When an argument name is used in the *statements* of a function, it is replaced by a copy of the corresponding argument value.

For example, the following is a simple function that takes a single numeric argument *N* and returns a random integer between 1 and *N* (inclusive):

```
function random(N) {
    return (int(N * rand() + 1))
}
```

Passing an Array to a Function

When an array is passed as an argument to a function, it is passed *by reference*. This means that the function works with the actual array, not with a copy. Anything that the function does to the array has an effect on the original array. **split** is a built-in function that takes an array as an argument.

split(*string*,*array***)**

split breaks up *string* into fields, and assigns each of the fields to an element of *array*. The first field is assigned to *array*[1], the next to *array*[2], and so on. Fields are assumed to be separated with the field separator string FS. If you want to use a different field separator string, you can use:

split(*string*,*array*,*fsstring*)

where *fsstring* is the field separator string you want to use instead of FS. The result of **split** is the number of fields that *string* contained.

Note: **split** actually changes the elements of array. When an array is passed to a function, the function may change the array elements.

The Getline Function

The **getline** function reads input from the current data file or from a different file.

Running System Commands

You can run commands with the **system** function:

```
system("command line")
```

runs the given command line: For example:

```
system("cd XYZ")
```

runs a **cd** command to change the working directory.

Controlling awk Output

By default, **awk** output is written to your workstation screen. You can save the output of an **awk** program in a file by using *output redirection*. To do this, put:

```
>filename
```

on the end of any **awk** command line. For example:

```
awk –f progfile datafile >outfile
```

writes all the output from the **awk** program to a file named **outfile**. In this case, the output does not appear on the workstation screen.

Formatting the Output

The output of the program:

```
$1 == "Jim" { print "$", $4/52 }
```

is:

```
$ 1.92308
$ 0.192308
$ 1.34615
```

This output shows the amount of money per week that Jim spent on his hobbies. However, money amounts usually have only two digits after the decimal point. How can you change the program to make the money amounts appear more normal? The answer is to use the **printf** action instead of **print**. This lets you specify the *format* in which **awk** prints the output.

A **printf** action looks like this:

```
{ printf format-string, value, value, ... }
```

The *format-string* indicates the output format. The *values* are the data to be printed.

A format string contains two kinds of items:

- *Normal characters,* which are just printed out as is
- *Placeholders,* which **awk** replaces with values given later in the **printf** action

As an example, try running the following program on the **hobbies** file:

```
$2 == "bridge" { printf "%5s plays bridge\n", $1 }
```

awk prints:

```
  Jim plays bridge
Linda plays bridge
 Lori plays bridge
```

The format string:

```
"%5s plays bridge\n"
```

has one placeholder: `%5s`. When **printf** prints its output, replacing the placeholder with the value `$1`, which is the first field of the record being examined. The rest of the format string is just printed out as is.

Note: The format string ends in `\n`; for more information, see “Escape Sequences” on page 343.

Placeholders

The form of the placeholder `%5s` tells **awk** how to print the associated value. All placeholders begin with `%` and end in a letter. The following are some of the most common letters used in placeholders:

c If the associated value is an integer, **printf** prints the character in the native character set that has that integer value; if the associated value is a string, **printf** prints the first character of the string.

d An integer in decimal form (base 10).

e A floating-point number in scientific notation, as in `-d.ddd dddE+dd`.

f A floating-point number in conventional form, as in `-ddd.ddd ddd`.

g A floating-point number in either `e` or `f` form, whichever is shorter; also, nonsignificant zeros are not printed.

o An unsigned integer in octal form (base 8).

s A string.

x An unsigned integer in hexadecimal form (base 16).

For example, the format string:

```
"%s %d\n"
```

contains two placeholders: `%s` represents a string, and `%d` represents a decimal integer.

Between the `%` and the letter at the end of the placeholder, you can put additional information. If you put an integer, as in `%5s`, the number is used as a *width*. **awk** prints the corresponding value using (at least) the given number of characters. Therefore in:

```
$2 == "bridge" { printf "%5s plays bridge\n", $1 }
```

the value of the string `$1` replaces the placeholder `%5s` and is always printed using five characters. The output is therefore:

```
  Jim plays bridge
Linda plays bridge
 Lori plays bridge
```

as shown before. If you just write:

```
$2 == "bridge" { printf "%s plays bridge\n", $1 }
```

without the `5`, the output is:

```
Jim plays bridge
Linda plays bridge
Lori plays bridge
```

If no *width* is given, **awk** prints values using the smallest number of characters possible.

awk also lets you put a minus sign (–) in front of the number in the width position. The amount of output space is the same, but the information is left-justified. For example:

```
$2 == "bridge" { printf "%-5s plays bridge\n", $1 }
```

prints:

```
Jim   plays bridge
Linda plays bridge
Lori  plays bridge
```

A placeholder for a floating-point number can also contain a *precision*. You can write this as a dot (decimal point) followed by an integer. Specifying a precision tells **printf** how many digits to print after the decimal point in a floating-point number. For example, in:

```
$1 == "John" { printf "$%.2f on %s\n", $4 * 1.05, $2 }
```

the placeholder `%.2f` indicates that **printf** is to print all floating-point numbers with two digits after the decimal point. The output of this program is:

```
$105.00 on role playing
$31.50 on jogging
```

For good-looking output, you might specify both a width and a precision. For example, the program:

```
$1 == "John" { printf "$%6.2f on %s\n", $4 * 1.05, $2 }
```

prints the following:

```
$105.00 on role playing
$ 31.50 on jogging
```

`%6.2f` indicates that the corresponding floating-point value should be printed with a width of six characters, with two characters after the decimal point.

Here are a few more **awk** programs that work on the **hobbies** file. Predict what each prints and run them to see if your prediction is right:

```
(a)  { printf "%6s %s\n", $1, $2 }
(b)  { printf "%20s: %2d hours/week\n", $2, $3 }
(c)  $1=="Katie" { printf "%20s: $%6.2f\n",$2,$4 }
```

Escape Sequences

All the format strings shown so far have ended in `\n`. This kind of construct is called an *escape sequence*. All escape sequences are made from a backslash character (\) followed by one to three other characters.

Escape sequences are used inside strings, not just those for **printf**, to represent special characters. In particular, the `\n` escape sequence represents the newline character. A `\n` in a **printf** format string tells **awk** to start printing output at the beginning of a newline.

The following list shows escape sequences that can be used in **awk** strings:

Escape	ASCII Character
`\a`	Audible bell
`\b`	Backspace
`\f`	Formfeed
`\n`	Newline
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\`*ooo*	ASCII character, octal *ooo*
`\x`*dd*	Hexadecimal value *dd*
`\"`	Quote
`\`*c*	Any other character *c*

Appendix E. Code Page Conversion when the Shell and MVS Have Different Locales

A *code page* for a specific character set determines the graphic character produced for each hexadecimal encoding. The code page used is determined by the programs and national languages being used.

If the shell is using a locale generated with code pages IBM-1047, IBM-1027, or IBM-939, an application programmer needs to be concerned about "variant" characters in the POSIX portable character set whose encoding may vary from other EBCDIC code pages:

- Right brace (})
- Left brace ({)
- Backslash (\)
- Right square bracket (])
- Left square bracket ([)
- Circumflex (^)
- Tilde (~)
- Exclamation point (!)
- Pound sign (#)
- Vertical bar (|)
- Dollar sign ($)
- Commercial at-sign (@)
- Accent grave (`)

For example, the encodings for the square brackets do not match on code pages IBM-037 and IBM-1047:

- Left square bracket: [
- Right square bracket:]

Customizing the Variant Characters on Your Keyboard

Assuming that you are not using an APL character set, on many programmable workstations you can customize your keys so that you have hexadecimal encodings for the variant characters that match the shell-supported code pages. For example, for those brackets the compatible encodings would be:

- X'AD' for a left square bracket ([)
- X'BD' for a right square bracket (])

Using the CONVERT Option on the OMVS Command

The OMVS command has a CONVERT option that lets you specify a conversion table for converting between code pages. The table you want to specify depends on the code pages you are using in MVS and in the shell. For example, if you are using code page IBM-037 in MVS and code page IBM-1047 in the shell, specify the following when you enter the OMVS command:

```
OMVS CONVERT((BPXFX111))
```

For more information, see the OMVS command description in *OS/390 UNIX System Services Command Reference*.

When Do You Need to Convert between Code Pages?

If you are using code page IBM-037 in MVS and the shell is using code page IBM-1047, you need to convert from one code page to another when:

- Transferring files between a workstation and the file system.
- Copying data between MVS data sets and the file system.
- Passing JCL pathname data to OS/390 UNIX programs—unless you restrict yourself to characters in the POSIX portable filename character set.
- Passing JCL parameters and pathnames to a shell invoked from a batch program—unless you restrict yourself to characters in the POSIX portable filename character set.
- Converting between ASCII and EBCDIC when using the **pax** utility.

Methods for Converting Data

There are several methods for converting data to or from a shell-supported code page:

- To convert data you are typing at a 3270 terminal, you specify a conversion table *other than* BPXFX100 (the null conversion table) with the OMVS command. The data you type at your workstation when you are working in the shell is converted to a shell-supported code page.
- To convert data between code pages IBM-037 and IBM-1047 when you are moving the data to or from the hierarchical file system, you can use the CONVERT option on the OPUT, OGET, and OCOPY commands.
- To convert doublebyte or singlebyte data to a selected code page while you are working in MVS, use the OS/390 c/c++ **iconv** utility. For information on how to use this utility, see *OS/390 C/C++ User's Guide*.
- To convert doublebyte or singlebyte data to a selected code page while you are working in the shell, use the shell **iconv** utility.

The POSIX Portable Filename Character Set

To simplify conversion requirements, use the POSIX portable filename character set when naming your files:

Uppercase A to Z
Lowercase a to z
Numbers 0 to 9
Period (.)
Underscore (_)
Hyphen (-)

The POSIX Portable Character Set

The POSIX portable character set consists of

Uppercase A to Z
Lowercase a to z
Numbers 0 to 9

and these characters:

+	<	=	>
$	`	^	~
#	%	&	*

@	[	]	\
{	}	|	!
"	'	(	)
,	_	-	.
/	:	;	?

Appendix F. Escape Sequences for a 3270 Keyboard

When using a 3270 keyboard, you can use escape sequences to type:

- Portable characters not included on your keyboard. See “Escape Sequences for Portable Characters Not on Your Keyboard”.
- Control characters that are normally available on ASCII workstations, but not EBCDIC ones. See “Escape Sequences for Control Characters” on page 350.

In this appendix, the notation *EscChar* coupled with another letter (for example, <EscChar> m) indicates an escape sequence.

Escape Sequences for Portable Characters Not on Your Keyboard

If you do not have keys on your keyboard for the following portable characters, you can use an escape sequence to obtain them.

Table 13. Portable Characters: Escape Sequences

OpenEdition Escape Sequence	Character	ASCII Control Sequence
<EscChar> @ <EscChar> 0	<NUL>	Ctrl-@
<EscChar> g <EscChar> G	<alert>	Ctrl-G
<EscChar> h <EscChar> H	<backspace>	Ctrl-H
<EscChar> i <EscChar> I	<tab>	Ctrl-I
<EscChar> j <EscChar> J	<newline>	Ctrl-J
<EscChar> k <EscChar> K	<vertical-tab>	Ctrl-K
<EscChar> l <EscChar> L	<form-feed>	Ctrl-L
<EscChar> m <EscChar> M	<carriage-return>	Ctrl-M
<EscChar> (	[	[
<EscChar>)	]	]

<tab> character:

When writing makefiles for the **make** utility, you need to use a <tab> character. If you are using a shell editor, you can type a <tab> character as an <EscChar-I> sequence. After you press <Enter>, the tab displays as blank space.

If you are using the ISPF editor, you cannot type a <tab> character (ISPF handles only displayable characters). See “Typing Tabs in ISPF” on page 247 for information on how to enter a <tab> character when using the ISPF editor.

Escape Sequences for Control Characters

To obtain the following control characters, you must use an escape sequence.

Table 14. Control Characters: Escape Sequences

OpenEdition Escape Sequence	Character	ASCII Control Sequence
<EscChar> f <EscChar> F	<ACK>	Ctrl-F
<EscChar> x <EscChar> X	<CAN>	Ctrl-X
<EscChar> q <EscChar> Q	<DC1>	Ctrl-Q
<EscChar> r <EscChar> R	<DC2>	Ctrl-R
<EscChar> s <EscChar> S	<DC3>	Ctrl-S
<EscChar> t <EscChar> T	<DC4>	Ctrl-T
<EscChar> p <EscChar> P	<DLE>	Ctrl-P
<EscChar> y <EscChar> Y	<EM>	Ctrl-Y
<EscChar> e <EscChar> E	<ENQ>	Ctrl-E
<EscChar> d <EscChar> D	<EOT>	Ctrl-D
<EscChar> 2 <EscChar> [	<ESC>	Ctrl-[
<EscChar> w <EscChar> W	<ETB>	Ctrl-W
<EscChar> c <EscChar> C	<ETX>	Ctrl-C
<EscChar> 6 <EscChar> _	<IS1>	Ctrl-_
<EscChar> 5	<IS2>	Ctrl-^
<EscChar> 4 <EscChar>]	<IS3>	Ctrl-]
<EscChar> 3 <EscChar> \	<IS4>	Ctrl-\
<EscChar> u <EscChar> U	<NAK>	Ctrl-U
<EscChar> o <EscChar> O	<SI>	Ctrl-O
<EscChar> n <EscChar> N	<SO>	Ctrl-N
<EscChar> a <EscChar> A	<SOH>	Ctrl-A
<EscChar> b <EscChar> B	<STX>	Ctrl-B

Table 14. Control Characters: Escape Sequences (continued)

OpenEdition Escape Sequence	Character	ASCII Control Sequence
<EscChar> z <EscChar> Z	<SUB>	Ctrl-Z
<EscChar> v <EscChar> V	<SYN>	Ctrl-V

Escape Sequences Unique to a Conversion Table

Depending on the conversion table that you specify with the CONVERT keyword on the OMVS command, you may need to type a unique escape sequence to enter a character. This section shows how unique escape sequences are translated by each of the character conversion tables. The translations for escaped alphabetic characters (which are the same for all tables—these are Ctrl-A thru Ctrl-Z) are not shown in these tables.

BPXFX100 Conversion Table

This table shows the escape sequences for certain characters that may not be on your keyboard.

Table 15. Translation of Selected Escaped Characters (BPXFX100)

OpenEdition Escape Sequence	Character	ASCII Control Sequence
<EscChar> ? <EscChar> # <EscChar> 7	<DEL>	Ctrl-?
<EscChar> {	[	[
]	]	
<EscChar> ¬	<IS2>	Ctrl-^

BPXFX111 and BPXFX211 Conversion Tables

This table shows the escape sequences for certain characters that may not be on your keyboard.

Table 16. Translation of Selected Escaped Characters (BPXFX111 and BPXFX211)

OpenEdition Escape Sequence	Character	ASCII Control Sequence
<EscChar> ? <EscChar> # <EscChar> 7	<DEL>	Ctrl-?
<EscChar> {	[	[
]	]	
<EscChar> ¬	<IS2>	Ctrl-^

BPXFX437, BPXFX450, BPXFX471, BPXFX473, BPXFX477, BPXFX478, BPXFX480, BPXFX484, BPXFX485, BPXFX497 Conversion Tables

Conversion tables BPXFX437, BPXFX450, BPXFX471, BPXFX473, BPXFX477, BPXFX478, BPXFX480, BPXFX484, BPXFX485, and BPXFX497 have the following escape sequences for certain characters that may not be on your keyboard.

Table 17. Translation of Selected Escaped Characters. (BPXFX437, BPXFX450, BPXFX471, BPXFX473, BPXFX477, BPXFX478, BPXFX480, BPXFX484, BPXFX485, and BPXFX497)

OpenEdition Escape Sequence	Character	ASCII Control Sequence
<EscChar> ? <EscChar> 7	<DEL>	Ctrl-?
<EscChar> -	~	
<EscChar> %	@	
<EscChar> &	$	
<EscChar> ; <EscChar> !	|	
<EscChar> '	^	
<EscChar> =	#	
<EscChar> ″	`	
<EscChar> /	\	
<EscChar> :	!	
<EscChar> <	{	
<EscChar> >	}	
<EscChar> ^	<IS2>	Ctrl-^

Appendix G. Locale Objects, Source Files, and Charmaps

The OS/390 shell and utilities support the locales in Table 18. A *locale name* is the same as a *locale object name*. The suffix of the locale name, for example, IBM-277, indicates the code page that the locale is based on.

The *symbolic link* is a shortened name for the complete locale object name; You can use the *symbolic link* name when specifying a locale for an environment variable or with the **setlocale()** function. For example, you can specify

```
LANG=En_US
```

instead of

```
LANG=En_US.IBM-1047
```

The compiled locale object files are in the directory **/usr/lib/nls/locale**.

The compiled Two-Octet Universal Character Set (UCS-2) object files are in the directory **/usr/lib/nls/locale/uconvTable**.

Table 18. OS/390 UNIX Locale Objects

Locale Name (Locale Object Name)	Symbolic Link	Language	Country
C	none		
POSIX	none		
SAA	none		
Bg_BG.IBM-1025	Bg_BG	Bulgarian	Bulgaria
Cs_CZ.IBM-870	Cs_CZ	Czech	Czech Republic
Da_DK.IBM-277	none	Danish	Denmark
Da_DK.IBM-1047	Da_DK	Danish	Denmark
De_CH.IBM-500	none	German	Switzerland
De_CH.IBM-1047	De_CH	German	Switzerland
De_DE.IBM-273	none	German	Germany
De_DE.IBM-1047	De_DE	German	Germany
El_GR.IBM-875	El_GR	Ellinika	Greece
En_GB.IBM-285	none	English	United Kingdom
En_GB.IBM-1047	En_GB	English	United Kingdom
En_JP.IBM-1027	En_JP	English	Japan
En_US.IBM-037	none	English	United States
En_US.IBM-1047	En_US	English	United States
Es_ES.IBM-284	none	Spanish	Spain
Es_ES.IBM-1047	Es_ES	Spanish	Spain
Fi_FI.IBM-278	none	Finnish	Finland
Fi_FI.IBM-1047	Fi_FI	Finnish	Finland
Fr_BE.IBM-500	none	French	Belgium
Fr_BE.IBM-1047	Fr_BE	French	Belgium

Table 18. OS/390 UNIX Locale Objects (continued)

Locale Name (Locale Object Name)	Symbolic Link	Language	Country
Fr_CA.IBM-037	none	French	Canada
Fr_CA.IBM-1047	Fr_CA	French	Canada
Fr_CH.IBM-500	none	French	Switzerland
Fr_CH.IBM-1047	Fr_CH	French	Switzerland
Fr_FR.IBM-297	none	French	France
Fr_FR.IBM-1047	Fr_FR	French	France
Hr_HR.IBM-870	Hr_HR	Croatian	Croatia
Hu_HU.IBM-870	Hu_HU	Hungarian	Hungary
Is_IS.IBM-871	none	Iceland	Iceland
Is_IS.IBM-1047	Is_IS	Iceland	Iceland
It_IT.IBM-280	none	Italian	Italy
It_IT.IBM-1047	It_IT	Italian	Italy
Iw_IL.IBM-424	Iw_IL	Hebrew	Israel
Ja_JP.IBM-939	Ja_JP	Japanese	Japan
Ja_JP.IBM-1027	none	Japanese	Japan
Ko_KR.IBM-933	Ko_KR	Korean	Korea
Mk_MK.IBM-1025	Mk_MK	Macedonian	Macedonia
Nl_BE.IBM-500	none	Dutch	Belgium
Nl_BE.IBM-1047	Nl_BE	Dutch	Belgium
Nl_NL.IBM-037	none	Dutch	Netherlands
Nl_NL.IBM-1047	Nl_NL	Dutch	Netherlands
No_NO.IBM-277	none	Norwegian	Norway
No_NO.IBM-1047	No_NO	Norwegian	Norway
Pl_PL.IBM-870	Pl_PL	Polish	Poland
Pt_BR.IBM-037	none	Brazilian	Brazil
Pt_BR.IBM-1047	Pt_BR	Brazilian	Brazil
Pt_PT.IBM-037	none	Portuguese	Portugal
Pt_PT.IBM-1047	Pt_PT	Portuguese	Portugal
Ro_RO.IBM-870	Ro_RO	Romanian	Romania
Ru_RU.IBM-1025	Ru_RU	Russian	Russia
Sh_SP.IBM-870	Sh_SP	Serbian (Latin)	Serbia
Sk_SK.IBM-870	Sk_SK	Slovak	Slovakia
Sl_SI.IBM-870	Sl_SI	Slovenian	Slovenia
Sr_SP.IBM-1025	Sr_SP	Serbian (Cyrillic)	Serbia
Sv_SE.IBM-278	none	Swedish	Sweden
Sv_SE.IBM-1047	Sv_SE	Swedish	Sweden
Zh_CN.IBM-935	Zh_CN	Simplified Chinese	People's Republic of China

Table 18. OS/390 UNIX Locale Objects (continued)

Locale Name (Locale Object Name)	Symbolic Link	Language	Country
Zh_CN.IBM-1388	Zh_CN	Expanded Chinese	People's Republic of China
Zh_TW.IBM-937	Zh_TW	Traditional Chinese	Taiwan

Locale Source Files

The locale source definition files are in **/usr/lib/nls/localedef**.

The Two-Octet Universal Character Set (UCS-2) source definition files are in **/usr/lib/nls/locale/ucmap**.

The source file name combined with the code page name results in the name of the locale object.

Table 19. OS/390 UNIX Locale Source Files

Locale Source File Name	Code Page	Locale Object Name
Bg_BG	IBM-1025	Bg_BG.IBM-1025
Cs_CZ	IBM-870	Cs_CZ.IBM-870
Da_DK	IBM-277 IBM-1047	Da_DK.IBM-277 Da_DK.IBM-1047
De_CH	IBM-500 IBM-1047	De_CH.IBM-500 De_CH.IBM-1047
De_DE	IBM-273 IBM-1047	De_DE.IBM-273 De_DE.IBM-1047
El_GR	IBM-875	El_GR.IBM-875
En_GB	IBM-285 IBM-1047	En_GB.IBM-285 En_GB.IBM-1047
En_JP.IBM-1027	IBM-1027	En_JP.IBM-1027
En_US	IBM-037 IBM-1047	En_US.IBM-037 En_US.IBM-1047
Es_ES	IBM-284 IBM-1047	Es_ES.IBM-284 Es_ES.IBM-1047
Fi_FI	IBM-278 IBM-1047	Fi_FI.IBM-278 Fi_FI.IBM-1047
Fr_BE	IBM-500 IBM-1047	Fr_BE.IBM-500 Fr_BE.IBM-1047
Fr_CA	IBM-037 IBM-1047	Fr_CA.IBM-037 Fr_CA.IBM-1047
Fr_CH	IBM-500 IBM-1047	Fr_CH.IBM-500 Fr_CH.IBM-1047
Fr_FR	IBM-297 IBM-1047	Fr_FR.IBM-297 Fr_FR.IBM-1047
Hr_HR	IBM-870	Hr_HR.IBM-870
Hu_HU	IBM-870	Hu_HU.IBM-870

Table 19. OS/390 UNIX Locale Source Files (continued)

Locale Source File Name	Code Page	Locale Object Name
Is_IS	IBM-871 IBM-1047	Is_IS.IBM-871 Is_IS.IBM-1047
It_IT	IBM-280 IBM-1047	It_IT.IBM-280 It_IT.IBM-1047
Iw_IL	IBM-424	Iw_IL.IBM-424
Ja_JP.IBM-939	IBM-939	Ja_JP.IBM-939
Ja_JP.IBM-1027	IBM-1027	Ja_JP.IBM-1027
Ko_KR	IBM-933	Ko_KR.IBM-933
Mk_MK	IBM-1025	Mk_MK.IBM-1025
Nl_BE	IBM-500 IBM-1047	Nl_BE.IBM-500 Nl_BE.IBM-1047
Nl_NL	IBM-037 IBM-1047	Nl_NL.IBM-037 Nl_NL.IBM-1047
No_NO	IBM-277 IBM-1047	No_NO.IBM-277 No_NO.IBM-1047
Pl_PL	IBM-870	Pl_PL.IBM-870
Pt_BR	IBM-037	Pt_BR.IBM-037
Pt_PT	IBM-037 IBM-1047	Pt_PT.IBM-037 Pt_PT.IBM-1047
Ro_RO	IBM-870	Ro_RO.IBM-870
Ru_RU	IBM-1025	Ru_RU.IBM-1025
Sh_SP	IBM-870	Sh_SP.IBM-870
Sk_SK	IBM-870	Sk_SK.IBM-870
Sl_SI	IBM-870	Sl_SI.IBM-870
Sr_SP	IBM-1025	Sr_SP.IBM-1025
Sv_SE	IBM-278 IBM-1047	Sv_SE.IBM-278 Sv_SE.IBM-1047
Zh_CN	IBM-935	Zh_CN.IBM-935
Zh_CN	IBM-1388	Zh_CN.IBM-1388
Zh_TW	IBM-937	Zh_TW.IBM-937

Charmap Files

The charmap files are in **/usr/lib/nls/charmap**. The charmap file names are identical to code page names, for example, IBM-1047.

Appendix H. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, New York 10504-1785
USA

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2–31 Roppongi 3–chrome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION ″AS IS″ WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs

and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
2455 South Road
Poughkeepsie, NY 12601-5400
USA
Attention: Information Request

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming Interface

This book is intended to help the customer use an IBM OS/390 system with the OS/390 Shells and Utilities and the hierarchical file system (HFS).

This book also documents OS/390 UNIX General-use Programming Interface and Associated Guidance Information.

General-use programming interfaces allow the customer to write programs that obtain OS/390 UNIX services.

General-use Programming Interface and Associated Guidance Information is identified where it occurs, by an introductory statement to a chapter or section.

Trademarks

The following terms used in this book are trademarks of the IBM Corporation in the United States or other countries or both:

- ACF/VTAM
- ADSTAR
- AIX
- BookManager
- C++/MVS
- C/370
- C/MVS
- DFSMSdfp
- DFSMShsm
- DFSMS/MVS
- IBM
- IBMLink
- Language Environment
- Library Reader

- MVS/ESA
- OS/2
- OS/390
- RACF
- Risc System/6000
- RS/6000
- S/390
- SP
- System/370
- TalkLink
- VTAM

UNIX is a registered trademark of The Open Group in the United States and other countries.

The following terms may be trademarks or service marks of others:

ANSI	American National Standards Institute
DFS	Transarc Corporation
IEEE	Institute of Electrical and Electronics Engineers
Lotus	Lotus Development Corporation
MKS	Mortice Kern Systems
Notes	Lotus Development Corporation
OSF	Open Software Foundation, Inc.
POSIX	Portable Operating System Interface

Acknowledgments

InterOpen Shell and Utilities is a source code product providing POSIX.2 (Shell and Utilities) functions to the UNIX services offered with OS/390. InterOpen/POSIX Shell and Utilities is developed and licensed by Mortice Kern Systems (MKS) Inc. of Waterloo, Ontario, Canada.

The information contained in the glossary section and tagged by the word [POSIX] is copyrighted information of the Institute of Electrical and Electronics Engineers, Inc., extracted from IEEE Std 1003.1-1990, IEEE P1003.0, and IEEE P1003.2. This information was written within the context of these documents in their entirety. The IEEE takes no responsibility or liability for and will assume no liability for any damages resulting from the reader's misinterpretation of said information resulting from the placement and context in this publication. Information is reproduced with the permission of the IEEE.

Glossary

This glossary defines terms and abbreviations used in OS/390 UNIX System Services documentation. If you do not find the term you are looking for, refer to the index of the appropriate OS/390 UNIX System Services manual or view *IBM Dictionary of Computing*, located at: http://www.ibm.com/networking/nsg/nsgmain.htm

This glossary includes terms and definitions from:

- *IBM Dictionary of Computing* (New York: McGraw-Hill, 1994).
- *Information Technology—Portable Operating System Interface (POSIX),* from the POSIX series of standards for applications and user interfaces to open systems, copyrighted by the Institute of Electrical and Electronics Engineers (IEEE). Copies of all POSIX drafts and standards may be purchased from IEEE at 1-800-678-IEEE.
  - Definitions identified by *[POSIX.0]* are from *Part 0: Standards Project, Draft Guide to the POSIX Open System Environment,* P1003.0 Draft 15 (June 1992), an unapproved draft subject to change.
  - Definitions identified by *[POSIX.1]* are from *Part 1: System Application Program Interface (API) [C Language],* approved September 28, 1990, as IEEE Std 1003.1-1990 by the IEEE Standards Board, and adopted in 1990 as an International Standard (ISO/IEC 9945-1: 1990) by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC).
  - Definitions identified by *[POSIX.2]* are from *Part 2: Shell and Utilities,* P1003.2.
- *American National Standard Dictionary for Information Systems,* ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018. Definitions are identified by the symbol *[A]* after the definition.
- *Information Technology Vocabulary,* developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions from published sections of these vocabularies are identified by the symbol *[I]* after the definition. Definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol *[T]* after the definition, indicating that final agreement has not yet been reached among the participating national bodies of SC1.
- *Sixth Plenary Assembly Orange Book, Terms and Definitions* and working documents published by the International Telecommunication Union, Geneva, 1978. These are identified by the symbol *[CCITT/ITU]* after the definition.
- Open Software Foundation (OSF). These are identified by the symbol *[OSF]* after the definition. Copies of OSF documents may be obtained from Open Software Foundation, Inc., 11 Cambridge Center, Cambridge, MA 02142.

Sequence of Entries: For clarity and consistency of style, this glossary arranges the entries alphabetically on a letter-by-letter basis. In other words, only the letters of the alphabet are used to determine sequence; special characters and spaces between words are ignored.

Organization of Entries: Each entry consists of a single-word or multiple-word term or the abbreviation or acronym for a term, followed by a commentary. A commentary includes one or more items (definitions or references) and is organized as follows:

1. An item number, if the commentary contains two or more items.
2. A usage label, indicating the area of application of the term, for example, “In programming,” or “In TCP/IP.” Absence of a usage label implies that the term is generally applicable to the OS/390 UNIX System Services interface, to IBM, or to data processing.
3. A descriptive phrase, stating the basic meaning of the term. The descriptive phrase is assumed to be preceded by “the term is defined as ...” The part of speech being defined is indicated by the opening words of the descriptive phrase: “To ...” indicates a verb, and “Pertaining to ...” indicates a modifier. Any other wording indicates a noun or noun phrase.
4. Annotative sentences, providing additional or explanatory information.

5. References, directing the reader to other entries or items in the dictionary.
6. A source label—for example, *[A], [I], [T], [CCITT/ITU], [OSF], [POSIX.0], [POSIX.1],* or *[POSIX.2]*—that follows the definition and identifies the originator of the definition. Definitions without source labels are IBM definitions.

References: The following cross-references are used in this glossary:

Contrast with. This refers to a term that has an opposed or substantively different meaning.
Synonym for. This indicates that the term has the same meaning as a preferred term, which is defined in its proper place in the glossary.
Synonymous with. This is a backward reference from a defined term to all other terms that have the same meaning.
See. This refers you to multiple-word terms that have the same last word.
See also. This refers the reader to related terms that have a related, but not synonymous, meaning.
Deprecated term for or **Deprecated abbreviation for.** This indicates that the term or abbreviation should not be used. It refers to a preferred term, which is defined in its proper place in the glossary.

Selection of Terms: A term is a word or group of words to be defined. In this glossary, the singular form of the noun and the infinitive form of the verb are the terms most often selected to be defined. If the term may be abbreviated, the abbreviation is given in parentheses immediately following the term. The abbreviation is also defined in its proper place in the glossary.

A

absolute pathname. (1) A pathname beginning with a slash. The predecessor of the first filename in the pathname is taken to be the root directory of the process. [POSIX.1] (2) The name of any directory or file expressed as a string of directories and files beginning with the root directory. (3) The name of a shared library key that begins with a slash. The system does not append any directories to the name of the shared library key, and it attempts to open a file with that name. [OSF] (4) See also *pathname, relative pathname*.

access permission. A group of designations that determine who can access a particular file and how the user can access the file.

ACF/VTAM. Advanced Communications Function for the Virtual Telecommunications Access Method.

action. In **awk**, **lex**, and **yacc**, a C language program fragment that defines what the program does when it recognizes input. [OSF]

address space. (1) The memory locations that can be referred to by a process. [POSIX.1] (2) The code, stack, and data that are accessible by a process. (3) The complete range of addresses that is available to a programmer. (4) The area of virtual storage available for a program. (5) See *forked address space, kernel address space, user address space*.

Advanced Communications Function for the Virtual Telecommunications Access Method (ACF/VTAM). A licensed program that provides single-domain network capability and, optionally, multiple-domain capability. It controls communication and the flow of data in an SNA network between terminals and application programs running under VSE and OS/VS2.

alert. (1) To cause the user's terminal to give some audible or visual indication that an error or some other event has occurred. When the standard output is directed to the terminal device, the method of alerting the terminal user is unspecified. When the standard output is not directed to the terminal device, the alert shall be accomplished by writing the <alert> character to standard output (unless the utility description indicates that the use of standard output produces undefined results in this case). [POSIX.2] (2) A character in the output stream that indicates that a terminal should alert its user via a visual or audible notification. The <alert> is the character designated by "\a" in the C language binding. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the alert function. [POSIX.2]

alias. (1) An alternate name for a shell command. An alias for a command can be defined with options different from those for the command itself. An alias can be a convenient shorthand for a complicated command line, such as a pipeline. (2) An alternate name for a user, system, or file that can be used in place of the real name of the object. [OSF] An alias may be defined to change the default options given to a particular command (for example, **pg**), or may be a shorthand for a more complicated command line (for example, a pipeline). (3) An alternate label. For example, a label and one or more aliases may be used to refer to the same data element or point in a computer program. [A] (4) An alternate name for a member of a partitioned data set. (5) Synonymous with nickname.

allocate. To assign a resource, such as a disk file, to a specific task. Contrast with *deallocate.*

American National Standard Code for Information Interchange (ASCII). The code developed by ANSI for information interchange among data processing systems, data communications systems, and associated equipment. The ASCII character set consists of 7-bit control characters and symbolic characters. [OSF] See also *extended binary-coded decimal interchange code (EBCDIC)*.

append. The action that causes data to be added to the end of existing data. [OSF]

appropriate privileges. Having sufficient authority to perform a requested function. If applications have been assigned an UID of 0, which indicates superuser authority, they can call any service. Security product profiles can also grant or restrict a user's access to certain functions.

archive. (1) To store programs and data for safekeeping. [OSF] (2) A copy of one or more files or a copy of a database that is saved in case the original data is damaged or lost. (3) Synonymous with backup, backup copy.

argument. (1) An independent variable. [I][A] (2) Any value of an independent variable, for example, a search key; a number identifying the location of an item in a table. [I][A] (3) A parameter passed between a calling and a called program. (4) A parameter passed to a utility as the equivalent of a single string in the *argv* array created by one of the POSIX.1 **exec** functions. An argument is one of the options, option-arguments, or operands following the command name. [POSIX.2] (5) Numbers, letters, or words that expand or change the way a command works. [OSF] (6) See *command-line argument*.

argument list. A string of arguments.

array. (1) A number of items stored together, which a user can quickly retrieve by supplying the correct index. Both **awk** and the OS/390 shells support arrays. (2) A variable that contains an ordered group of data objects. All objects in an array have the same data type. [OSF] (3) An arrangement of data in one or more dimensions, such as a list, table, or multidimensional arrangement of items. (4) In programming languages, an aggregate that consists of data objects, with identical attributes, each of which may be uniquely referenced by subscripting. [I]

ASCII. American National Standard Code for Information Interchange.

asynchronous device. A device using asynchronous data transmission.

asynchronous transmission. Data transmission for a character or block of characters, in which the bits are sent one after another, with a start bit and a stop bit that mark the beginning and end of a data unit. The interval between data units can be uneven—another character might follow immediately or there might be an idle interval before another character is sent. Contrast with *synchronous transmission*.

awk. A file processing language that is well suited to data manipulation and retrieval of information from text files. **awk** is named for its creators, Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan.

B

background. (1) In multiprogramming, the conditions under which low-priority programs are executed. (2) A mode of program execution in which the shell does not wait for program completion before prompting the user for another command. [OSF] Synonym for *background job*. (3) Contrast with *foreground.*

background job. An MVS job that the MVS initiator selects from a queue and starts running. Synonymous with background, batch job.

background process. (1) A process that is a member of a background process group. [POSIX.1] (2) A process that does not require operator intervention but can be run by the computer while the workstation is used to do other work. (3) A process that an interactive user starts running and that cannot interact with the user. An interactive user can move a background process to the foreground. (4) See also *background process group, background processing*.

background process group. Any process group, other than a foreground process group, that is a member of a session that has established a connection with a controlling terminal. [POSIX.1]

backslash. The character "\"—also known as a *reverse solidus.* [POSIX.2] The backslash enables a user to escape the special meaning of a character. That is, typing a backslash before a character tells the system to ignore any special meaning the character might have.

backspace. A character that normally causes printing (or displaying) to occur one column position previous to the position about to be printed. The <backspace> is the character designated by "\b" in the C language binding. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the backspace function. The <backspace> character defined here is not necessarily the ERASE special character defined in POSIX.1. [POSIX.2]

basename. (1) The final, or only, filename in a pathname. [POSIX.2] The basename is the part of a pathname that remains after all directory names are removed. For example, for the pathname **dir1/dir2/file.c**, the basename is **file.c**. (2) The last element to the right of a full pathname. (3) A filename specified without its parent directories.

basic regular expression. A pattern (sequence of characters or symbols) constructed according to the rules defined in POSIX.2 2.8.3. [POSIX.2]

batch job. (1) An MVS job that the MVS initiator selects from a queue and starts running. The UNIX equivalent is a *batch process*. (2) A job that is grouped with other jobs as input to a computing system. (3) Synonymous with background job, batch, batched job.

batch processing. (1) The processing of data or the accomplishment of jobs accumulated in advance in such a manner that the user cannot further influence processing while it is in progress. [I][A] (2) The processing of data accumulated over a period of time. [A] (3) The technique of executing a set of computer programs such that each is completed before the next program of the set is started. [A] (4) In realtime systems, the processing of related transactions that have been grouped together. (5) Loosely, the execution of computer programs serially. [A] (6) The sequential input of computer programs or data. (7) A processing method in which a program or programs process records with little or no operator action in a background process. [OSF] (8) Contrast with *interactive processing*. (9) See also *background processing*.

binary data. (1) Any data not intended for direct human reading. Binary data may contain "unprintable" characters, outside the range of text characters. (2) A type of data consisting of numeric values stored in bit patterns of `0`s and `1`s. Binary data can cause a large number to be placed in a smaller space of storage.

binary file. A file that contains codes that are not part of the character set. Binary files utilize all 256 possible values for each byte in the file. See also *text file*.

blank. One of the characters that belong to the *blank* character class as defined via the LC_CTYPE category in the current locale. In the POSIX locale, a <blank> is either a <tab> or a <space>. [POSIX.2]

Bourne Shell. The shell based on UNIX V. See also *shell*.

braces. The characters "{" (*left brace*) and "}" (*right brace*), also known as *curly braces*. When used in the phrase "enclosed in (curly) braces," the symbol "{" immediately precedes the object to be enclosed, and "}" immediately follows it. Synonymous with <left-brace> and <right-brace>. [POSIX.2] Synonymous with curly brackets.

brackets. The characters "[" (*left bracket*) and "]" (*right bracket*), also known as *square brackets*. When used in the phrase "enclosed in (square) brackets," the symbol "[" immediately precedes the object to be enclosed, and "]" immediately follows it. Synonymous with <left-square-bracket> and <right-square-bracket>. [POSIX.2]

C

callable service. (1) A request by an active process for a service by the system kernel. (2) A call to receive OS/390 UNIX System Services. (3) Synonymous with *syscall*, *system call*.

canonical mode. A **tty** input processing mode where input is collected and processed one line at a time. [OSF] There are three CLISTs associated with canonical mode drivers: store output to a terminal, store raw input, store cooked data. I/O processing is asymmetric. Synonymous with cooked mode, line mode. Contrast with *noncanonical mode*.

carriage return (CR). (1) A keystroke generally indicating the end of a command line. (2) In text data, the action that indicates to continue printing at the left margin of the next line. [OSF] (3) A character that in the output stream causes printing to start at the beginning of the same physical line in which the <carriage return> occurred. The <carriage return> is the character designated by the "\r" in the C language binding. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the beginning of the line. [POSIX.2]

case-sensitive. Pertaining to the ability to distinguish between uppercase and lowercase letters.

catalog. (1) A directory of files and libraries, with reference to their locations. [I][A] (2) To enter information about a file or a library into a catalog. [I][A] (3) The collection of all data set indexes that are used by the control program to locate a volume containing a specific data set.

character. (1) A letter, digit, or other symbol. (2) A letter, digit, or other symbol that is used as part of the organization, control, or representation of data. A character is often in the form of a spatial arrangement of adjacent or connected strokes. [A] (3) A sequence of one or more bytes representing a single graphic symbol. The term corresponds in the C Standard to the term *multibyte character,* noting that a single-byte character is a special case of a multibyte character. In POSIX, however, a character has no necessary relationship with storage space, and *byte* is used when storage space is discussed. [POSIX.1] (4) A member of a set of elements that is used for the representation, organization, or control of data. Characters may be letters, digits, punctuation marks, or other symbols. [T]

character class. (1) A named set of characters sharing an attribute associated with the name of the class. The classes and the characters that they contain are dependent on the value of the LC_CTYPE category in the current locale. [POSIX.2] (2) Ranges of characters that match a single character in the input

stream. [OSF] (3) A set of characters enclosed in sequence, or square, brackets. [OSF]

character set. (1) A defined collection of characters. (2) All the valid characters for a programming language or for a computer system. (3) A group of characters used for a specific reason—for example, the set of characters a printer can print or a keyboard can support.

character special file. (1) A special file that provides access to an input or output device. The character interface is used for devices that do not use block I/O. (2) A file that refers to a device. One specific type of character special file is a terminal device file. Other character special files have no structure defined by POSIX.1, and their use is unspecified by POSIX.1. [POSIX.1] The only character special file supported by the OS/390 UNIX implementation is the pseudo–TTY, or pseudoterminal.

child process. A process created as a result of a fork. The child process receives a copy of the parent's storage and inherits open files. Execution in the child continues at the instruction following the fork. Contrast with *parent process*. See also *fork, process.*

class. (1) Any category to which things are arranged or defined. (2) A regular expression that matches any one of a set of characters.

CLIST. Command list.

code page. (1) A table showing codes assigned to character sets. (2) An assignment of graphic characters and control function meanings to all code points. (3) Arrays of code points representing characters that establish ordinal sequence (numeric order) of characters. [OSF] (4) A particular assignment of hexadecimal identifiers to graphic elements. (5) Synonymous with code set. (6) See also *code point, extended character*.

code point. A 1-byte code representing one of 256 potential characters.

collating element. The smallest entity used to determine the logical ordering of strings. A collating element shall consist of either a single character, or two or more characters collating as a single entity. The value of the LC_COLLATE category in the current locale determines the current set of collating elements. See also *collation sequence.* [POSIX.2]

collating sequence. Synonym for *collation sequence*.

collation sequence. (1) A specified arrangement used in sequencing. [I][A] (2) The sequence in which characters are ordered within the computer for sorting, combining or comparing. (3) The relative order of collating elements as determined by the setting of the LC_COLLATE category in the current locale. The character collation sequence defines the relative order of all collating elements such that each element occupies a unique position in the sequence. This is also the default sorting sequence. Multilevel sorting is accomplished by assigning elements one or more collation weights, up to the limit {COLL_WEIGHTS_MAX}. On each level, elements may be given the same weight or can be omitted from the sequence. Strings that collate equally using the first assigned weight (primary ordering) are then compared using the next assigned weight (secondary ordering), and so on. See also *equivalence class*. [POSIX.2] (4) Synonymous with collating sequence.

column. The vertical arrangement of characters or other expressions. [A] Contrast with *row.*

command alias. An abbreviation of a long command line or a new name for a command. [OSF]

command history. A feature that stores commands and allows you to edit and reuse them. [OSF]

command interpreter. A program that reads the commands that you type in and then executes them. When you are typing commands into the computer, you are actually typing input to the command interpreter. The interpreter then decides how to perform the commands that you have typed. The shell is an example of a command interpreter. Synonymous with command language interpreter.

command line. (1) The area of the screen where commands are displayed as they are typed. [OSF] (2) The position after the arrow (===>) at the top or bottom of the panel where the user can type a command. (3) A unit of input to the shell, consisting of a program name and possibly command line arguments, separated by white space and terminated by a newline.

command-line argument. A part of a command line, delimited by white space. Arguments are used to specify detailed behavior to a program. They are usually either command line options selecting variations in program operation, or pathnames of files to be processed.

command list (CLIST). (1) A data set in which commands and possibly subcommands and data are stored for subsequent execution. (2) A sequential list of commands, control statements, or both that is assigned a name and executed when that name is invoked.

command mode. (1) A state of a system or device in which the user can enter commands. (2) In an editing session, the mode wherein the editor is waiting for the user to enter a command.

command name. (1) The first term in a command. It requests a specific action, and it is usually followed by operands. (2) The first or principal term in a command. A command name does not include parameters, arguments, flags, or other operands. [OSF]

command processor. A program executed to perform an operation specified by a command.

command substitution. The ability to capture the output of any command as an argument to another command by placing that command line within grave accents (` `). The shell first runs the command or commands enclosed within the grave accents and then replaces the whole expression, including grave accents, with their output. This feature is often used in assignment statements.

comment. An English language description of what a program does. In the shell and in **awk**, a comment is introduced with a # character and ends at the end of the line.

concatenate. (1) To link together. (2) To join two character strings.

console. (1) The main system display station. [OSF] (2) A device name associated with the main system display station. [OSF]

controlling process. The session leader that established the connection to the controlling terminal. If the terminal subsequently ceases to be a controlling terminal for this session, the session leader shall cease to be the controlling process. [POSIX.1]

controlling terminal. (1) An active terminal at which a user is authorized to enter commands that affect system operation. The controlling terminal for any process normally is the active terminal from which the process group for that process was started. A terminal can have no more than one controlling process group and a process group can have no more than one controlling terminal. The controlling process group receives certain interrupt signals from the controlling terminal. [OSF] (2) A terminal that is associated with a session. Each session may have at most one controlling terminal associated with it, and a controlling terminal is associated with exactly one session. Certain input sequences from the controlling terminal cause signals to be sent to all processes in the process group associated with the controlling terminal. [POSIX.1]

conversion. (1) The process of changing from one form of representation to another—for example, to change from decimal representation to binary representation; or to change from ASCII to EBCDIC. (2) A change in the type of value. For example, in some programming languages when you add values having different data types, the compiler converts both values to the same form before adding them. (3) In programming languages, the transformation between values that represent the same data item but belong to different data types. Information may be lost due to conversion since accuracy of data representation varies among different data types. [I] (4) The process of changing from one method of data processing to another or from one data processing system to another. (5) Sometimes referred to as translation.

conversion table. (1) A table used to replace one or more characters with alternative characters—for example, to convert characters representing an event to those representing a procedure call, characters of a code set to those of another code set, or characters representing a relocated address to those representing an absolute address. (2) A table that maps virtual addresses with real addresses. (3) A table that specifies the mapping of events or event sequences to procedure names. (4) Synonymous with translate table, translation table.

country–extended code page (CECP). An 8-bit code page that has a 93-character set on its nationally standardized code points but is extended to the multilingual character set for the national languages of some European countries.

current directory. The directory that is active and can be displayed with the **pwd** command. Synonymous with current working directory. [OSF]

current file. The file being edited. If multiple windows are in use, the current file is the file containing the cursor. [OSF]

current line. The line on which the cursor is located. [OSF]

current record. (1) The record pointed to by the current line pointer. (2) The record that is currently available to the program.

current working directory. (1) The directory a user is working with. (2) Synonym for *current directory*. (3) Synonym for *working directory.*

D

daemon. A long-lived process that runs unattended to perform continuous or periodic systemwide functions, such as network control. Some daemons are triggered automatically to perform their task; others operate periodically. An example is the **cron** daemon, which periodically performs the tasks listed in the **crontab** file. The MVS equivalent is a *started task*.

data definition. A program statement that describes the features of, specifies relationships of, or establishes the context of, data. [A] A data definition can also provide an initial value. Definitions appear outside a function or at the beginning of a block statement.

data definition name (ddname). (1) The name of a data definition (DD) statement that corresponds to a data control block that contains the same name. (2) The symbolic representation for a name placed in the name field of a data definition (DD) statement.

data set. (1) The major unit of data storage and retrieval in the operating system, consisting of a collection of data in one of several prescribed arrangements and described by control information to which the system has access. (2) The major unit of data storage and retrieval in an MVS system. A hierarchical file system is stored in an HFS data set.

data structure. The syntactic structure of symbolic expressions and their storage characteristics. [T]

ddname. Data definition name.

DD statement. Data definition statement.

deallocate. (1) To release a resource that is assigned to a specific task. (2) A request to remove the allocation of the specified conversation from the local transaction program. [OSF] (3) Contrast with *allocate.*

default directory. The directory name supplied by the operating system if none is specified.

descriptor. An unsigned integer that a UNIX system uses to identify an object supported by the kernel. Descriptors can represent files, pipes, sockets, and other I/O streams. They are created, acted on, and deallocated by system calls specific to the object. [OSF]

DFSMS. Data Facility System-Managed Storage.

directory. (1) A type of file containing the names and controlling information for other files or other directories. (2) A construct for organizing computer files. As files are analogous to folders that hold information, a directory is analogous to a drawer that can hold a number of folders. Directories can also contain subdirectories, which can contain subdirectories of their own. (3) A file that contains directory entries. No two directory entries in the same directory can have the same name. [POSIX.1] (4) A file that points to files and to other directories. (5) An index used by a control program to locate blocks of data that are stored in separate areas of a data set in direct access storage.

directory entry. An object that associates a filename with a file. Several directory entries can associate names with the same file. Synonymous with link. [POSIX.1]

dot. (1) A symbol (**.**) that indicates the current directory in a relative pathname. [OSF] (2) The filename consisting of a single dot character (**.**). This filename refers to the directory specified by its predecessor. See also *period.* [POSIX.1]

dot-dot. (1) A symbol (**..**) in a relative pathname that indicates the parent directory. [OSF] (2) The filename consisting solely of two dot characters (**..**). This filename refers to the parent directory of its predecessor directory. As a special case, in the root directory, dot-dot may refer to the root directory itself. [POSIX.1]

double-quote. The character " —also known as *quotation-mark*. [POSIX.2]

dub. To make an address space known to the UNIX kernel. Once dubbed, an address space is also considered to be a UNIX process. Address spaces created by **fork()** are automatically dubbed when they are created; other address spaces become dubbed if they invoke a kernel service. Dubbing also applies to OS/390 tasks. A dubbed task is considered a “thread.” Tasks created by pthread_create are automatically dubbed threads; other tasks are dubbed if they invoke a kernel service.

dump. (1) To write at a particular instant the contents of storage, or part of storage, onto another data medium for the purpose of safeguarding or debugging the data. [T] (2) To copy data in a readable format from central or auxiliary storage onto an external medium such as tape or printer. (3) To copy the contents of all or part of virtual storage for the purpose of collecting error information. (4) To copy all or part of the contents of an auxiliary storage device for later restoration of the data to an auxiliary storage device of the same type or another type.

E

EBCDIC. Extended binary-coded decimal interchange code.

echo. (1) In data communication, a reflected signal on a communication channel. (2) In word processing, to print or display each character or line as it is keyed in.

effective root directory. The point where a system starts when searching for a file. The pathname of the effective root directory begins with a slash (/).

effective user ID. (1) The current user ID, but not necessarily the user’s login ID. For example, a user logged in under a login ID may change to another user’s ID. The ID to which the user changes becomes the effective user ID until the user switches back to the original login ID. [OSF] (2) An attribute of a process that is used in determining various permissions, including file access permissions. This value is subject to change during the process lifetime. See also *user ID.* [POSIX.1]

element. (1) In a set, an object, entity, or concept having the properties that define a set. [I][A] Synonymous with member. (2) The smallest unit of data in a table or array. (3) The component of an array, subrange, enumeration, or set. (4) Any of the bits of a bit string, the octets of an octet string, or the octets by means of which the characters of a character string are represented. [OSF]

empty directory. A directory that contains, at most, directory entries for dot and dot-dot. [POSIX.1]

empty string. A character array whose first element is a null character. Synonymous with null string. [POSIX.1]

environment. (1) The settings for shell variables and paths set when the user logs in. These variables can be modified later by the user. [OSF] (2) A block of information passed ("exported") to a command when the command is invoked. This block contains a number of environment variables. The environment provides information that the program may use in its operation, in a form that relieves you of the need to specify it with every command. (3) The set of all factors that may affect how a program behaves. (4) A named collection of logical and physical resources used to support the operation of a function. (5) See also *environment variable*

environment variable. (1) A name associated with a string of characters, made available to the programs that you run. Some environment variables used by the OS/390 shell are **PATH**, **TMPDIR**, **COLUMNS**, and **LINES**. For example, the **TMPDIR** environment variable holds the name of a directory where shell commands are free to create temporary working files. (2) A variable that describes the operating environment of the process and typically includes information about the home directory, command search path, the terminal in use, and the current time zone (the **HOME**, **PATH**, **TERM**, and **TZ** variables, respectively). (3) An environment variable, the value of which is the name of a file that contains shell commands to customize a shell environment. When a user first runs the shell, the shell executes his or her profile. Then the shell runs the commands in the **ENV** file. (4) A variable included in the current software environment that is available to any called program that requests it.

equivalence class. (1) A grouping of characters or character strings that are considered equal for purposes of collation. For example, many languages place an uppercase character in the same equivalence class as its lowercase form, but some languages distinguish between accented and unaccented character forms for the purpose of collation. (2) A set of collating elements that collate equally on a particular weight level (primary, secondary, or following assigned collation weights). Elements in an equivalence class are typically elements that naturally group together, such as all accented letters based on the same base letter. The order of elements within an equivalence class is determined by the weights assigned on a subsequent level, if any. In regular expressions and pattern matching, only primary equivalence classes are recognized. [POSIX.2]

ERE. Extended regular expression.

escape character (ESC). (1) The control character in a text-control sequence that indicates the beginning of a sequence and the end of any preceding text. (2) In shell programming and **tty** programming, the \ (backslash) character, which indicates that the next character is not intended to have the special meaning normally assigned to it. (3) In general, a character that suppresses or selects a special meaning for one or more characters that follow. [OSF]

escape sequence. (1) A sequence of characters that begins with a \ (backslash) and is interpreted to have a special meaning to the shell. (2) A sequence sent to a terminal to perform actions such as moving the cursor, changing from normal to reverse video, and clearing the screen.

exec. To overlay the current process with another executable program. See also *fork*.

executable. See *executable file, executable program, executable statement*.

executable file. (1) A file suitable for execution. An executable file may be a program that has been compiled and link-edited, or it may be a shell script. (2) A file that contains programs or commands that perform operations on actions to be taken. (3) A regular file acceptable as a new process image file by the equivalent of the POSIX.1 **exec** family of functions, and thus usable as one form of a utility. The standard utilities described in POSIX.1 as compilers can produce executable files, but other unspecified methods of producing executable files may also be provided. The internal format of an executable file is unspecified, but a conforming application shall not assume an executable file is a text file. [POSIX.2] (4) See also *executable program*.

executable program. (1) A program in a form suitable for execution by a computer. The program can be an application or a shell script. An executable program is equivalent to an MVS load module. (2) A program that has been link-edited and can therefore be run in a processor. (3) A program that can be executed as a self-contained procedure. It consists of a main program and, optionally, one or more subprograms. (4) See also *executable file, load module*.

executable statement. A statement that causes an action to be taken by the program. For example, to calculate, to test conditions, or to alter normal sequential execution. [OSF]

export. To make a file system available to a client.

expression. (1) A representation of a value—for example, variables and constants appearing alone or in combination with operators. (2) In programming languages, a language construct for computing a value from one or more operands; for example, literals, identifiers, array references, and function calls. [I] (3) A configuration of signs. [A] (4) A group of constants or variables separated by operators that yields a single value. An expression can be arithmetic, relational, logical, or a character string. (5) An operand or a collection of operators and operands that yields a single result.

expression statement. An expression that ends with a **;** (semicolon). You can use an expression statement to assign the value of an expression to a variable or to call a function.

extended binary-coded decimal interchange code (EBCDIC). A coded character set consisting of 8-bit coded characters. [A] See also *American National Standard Code for Information Interchange (ASCII).*

extended character. A character other than a 7-bit ASCII character. An extended character can be a 1-byte code point with the eighth bit set (ordinal 128–255).

extended regular expression (ERE). A pattern (sequence of characters or symbols) constructed according to the rules defined in POSIX.2 2.8.4. [POSIX.2]

external link. A special type of symbolic link, a file that contains the name of an object that is outside of the hierarchical file system.

F

FIFO (first-in-first-out). A queuing technique in which the next item to be retrieved is the item that has been in the queue for the longest time. [A]

FIFO special file. (1) A type of file with the property that data written to such a file is read on a first-in-first-out basis. [POSIX.1] (2) A named permanent pipe that allows two unrelated processes to exchange information through a pipe connection. Synonymous with named pipe.

file. (1) A set of related records treated as a unit. (2) A sequence of records. If the file is located in internal storage, it is an internal file; if it is on an input/output device, it is an external file. [OSF] (3) A collection of related data that is stored and retrieved by an assigned name. [OSF] (4) Linear data that can be opened, written, read, and closed. A file can also contain information about the file, such as authorization information. The name used to obtain a file includes the directories in the path to the file. (5) Strings of characters with no additional structure. Structure is assumed only by the processing programs. Files can be located relative to the current directory or by an absolute pathname. (6) An object that can be written to, or read from, or both. A file has certain attributes, including access permissions and type. File types include regular file, character special file, block special file, FIFO special file, and directory. Other types of files may be defined by the implementation. [POSIX.1] In the OS/390 UNIX implementation, the file system does not support block special files, but it does support symbolic link files. (7) A collection of information or data that is organized by some method (relative, indexed, or serial, for example) and stored on a device such as a disk.

file descriptor. (1) A small unsigned integer that a UNIX system uses to identify a file. A file descriptor is created by a process through issuing an **open** system call for the filename. A file descriptor ceases to exist when it is no longer held by any process. [OSF] (2) A per-process unique, nonnegative integer used to identify an open file for the purpose of file access. [POSIX.1] (3) A small positive number used to identify an open file in I/O operations. By convention, certain file descriptors are used for the same purpose by all programs. (4) See also *standard error (stderr), standard input (stdin), standard output (stdout)*.

file hierarchy. A structure of all the files in the system whereby all the nonterminal nodes are directories and all the terminal nodes are any other type of file. Because multiple directory entries may refer to the same file, the hierarchy is properly described as a “directed graph”.

file mode. An object containing the file permission bits and other characteristics of a file. [POSIX.1]

filename. (1) A name assigned or declared for a file. (2) The name used by a program to identify a file. (3) A name consisting of 1 to {NAME_MAX} bytes used to name a file. The characters composing the name may be selected from the set of all character values excluding the slash character and the null character. The filenames dot and dot-dot have special meaning. Synonymous with pathname component. See also *dot, dot-dot.* [POSIX.1]

file owner. The user who has the highest level of access authority to a file, as defined by the file.

file system. (1) A collection of files and directories. (2) The collection of files and file management structures on a physical or logical mass storage device, such as a disk or disk partition. A single device can contain several file systems. (3) A mountable subtree of the directory hierarchy. [OSF] (4) A collection of files and certain of their attributes. A file system provides a name space for file serial numbers referring to those files. [POSIX.1] (5) See also *mountable file system*.

File Transfer Protocol (FTP). In TCP/IP, an application protocol used for transferring files to and from host computers. FTP requires a user ID and possibly a password to allow access to files on a remote host system. FTP assumes that the Transmission Control Protocol is the underlying protocol.

file type. One of the five possible types of files: ordinary file, directory, block device, character device, and first-in-first-out (FIFO or named pipe). See also *file.*

filter. (1) A command that reads standard input data, modifies the data, and sends it to standard output. A pipeline usually has several filters. (2) A program that reads in data (usually text), transforms it in some way, and writes out the result. Examples are a program that

reads in text, converts all letters to uppercase, and then writes out the result; a program that sorts its input and writes out the sorted data; and a program that reads in lines of data, eliminates duplicate lines, and writes out the result. (3) A device or program that separates data, signals, or materials in accordance with specified criteria. [A]

first-in-first-out (FIFO). A queuing technique in which the next item to be retrieved is the item that has been in the queue for the longest time. [A]

fixed-length record. A record having the same length as all other records with which it is logically or physically associated. Contrast with *variable-length record.*

flag. (1) A modifier that appears on a command line with the command name that defines the action of the command. [OSF] (2) An indicator or parameter that shows the setting of a switch. [OSF] (3) A variable indicating that a certain condition holds. [T] (4) A character that signals the occurrence of some condition, such as the end of a word. [A] (5) An internal indicator that describes a condition to the CPU. [OSF] Synonymous with condition code.

fold. To translate the lowercase characters of a character string into uppercase.

foreground. (1) A process that an interactive user starts running and that can interact with the user. An interactive user can move a foreground process to the background. (2) A mode of program execution in which the shell waits for the program specified on the command line to complete before responding to user input. (3) In multiprogramming, the environment in which high-priority programs are executed. (4) The interactive execution of programs and services. (5) The environment in which interactive programs are executed. Interactive processors reside in the foreground. (6) Contrast with *background.*

foreground process. (1) A process that is a member of a foreground process group. [POSIX.1] (2) A process that must run to completion before another command is issued to the shell. The foreground process is in the foreground process group.

foreground process group. A process group whose member processes have certain privileges, denied to processes in background process groups, when accessing their controlling terminal. Each session that has established a connection with a controlling terminal has exactly one process group of the session as the foreground process group of that controlling terminal. [POSIX.1] The foreground process group receives the signals generated by the terminal.

foreground processing. The execution of a computer program that preempts the use of computer facilities. [I][A]

fork. To create and start a child process. Forking creates a copy of the parent process, including open file descriptors. Forking is similar to attaching a subtask within a jobstep by issuing the ATTACH or ATTACHX macro in MVS.

forked address space. An address space created by a **fork()** function. A forked address space is perceived by MVS to be a batch job.

format. (1) A defined arrangement of such things as characters, fields, and lines, usually used for displays, printouts, or files. [OSF] (2) The pattern that determines how data is recorded. [OSF] (3) To arrange such things as characters, fields, and lines. (4) In programming languages, a language construct that specifies the representation, in character form, of data objects in a file. [I]

formatted file. A file displayed and arranged with particular characteristics, such as line spacing, headings, and number of characters and lines per page. [OSF] Contrast with *unformatted file*.

form-feed. A character that shall cause printing to start on the next page of an output device. The <form-feed> shall be the character designated by the "\f" in the C language binding. If <form-feed> is not the first character on an output line, the result is unspecified. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the next page. [POSIX.2]

full pathname. Synonym for *absolute pathname*.

function key. (1) A key that causes a specified sequence of operations to be performed when it is pressed. Generally used to refer to keys labeled <F*n*>, for example, <F1>. (2) A key that requests actions but does not display or print characters. This definition includes a key that normally produces a printed character, but produces a function instead when used with the code key. [OSF]

G

group. (1) A collection of users who can share access authorities for protected resources. [OSF] (2) A list of names that are known together by a single name. (3) A set of related records that have the same value for a particular field in all records. (4) A series of records logically joined together.

group ID (GID). (1) A unique number assigned to a group of related users. The GID can often be substituted in commands that take a group name as an argument. (2) A nonnegative integer, which can be contained in an object of type *gid_t,* that is used to identify a group of system users. Each system user is a member of at least one group. When the identity of a group is associated with a process, a group ID value is

referred to as a real group ID, an effective group ID, one of the (optional) supplementary group IDs, or an (optional) saved set-group-ID. [POSIX.1] (3) Synonymous with group number.

group name. A name that uniquely identifies a group of users to the system.

H

hard link. (1) A mechanism that allows the **ln** command to assign more than one name to a file. Both the new name and the file being linked must be in the same file system. [OSF] (2) The relationship between two directory entries that represent the same file; the result of an execution of the **ln** utility or the POSIX.1 **link()** function. [POSIX.2]

HFS data set. A hierarchical file system data set, which is used to store, and is essentially identified with, a file system.

HFS file. An object that exists within a mountable file system. Synonymous with POSIX file. See also *HFS data set*.

hierarchical file. See *file*.

high-order. Most significant; leftmost. [OSF]

history file. A file in which a record is kept of shell commands that are executed. The default history file is **.sh_history**.

home directory. (1) The current directory associated with the user at the time of login. [POSIX.2] (2) A directory associated with an individual user. (3) The user's current directory on login or after issuing the **cd** command with no argument.

I

index. (1) A list of the contents of a file or of a document, together with keys or references for locating the contents. [I][A] (2) A table used to locate records in an indexed sequential data set or an indexed file. (3) A table containing the key value and location of each record in an indexed file. [OSF] (4) A computer storage position or register whose contents identify a particular element in a set of elements. [OSF]

inherit. To copy resources or attributes from a parent to a child.

inode. The internal structure that describes the individual files in the operating system; there is one inode for each file. An inode contains the node, type, owner, access times, number of links, and location of a file. A table of inodes is stored near the beginning of a file system.

inode number. A number specifying a particular inode file in the file system. Synonymous with inumber. See also *inode*.

input redirection. The specification of an input source other than standard input/output.

interactive processing. A processing method in which each system user action causes response from the program or the system. Contrast with *batch processing*.

Interactive System Productivity Facility (ISPF). An IBM licensed program that serves as a full-screen editor and dialog manager. Used for writing application programs, it provides a means of generating standard screen panels and interactive dialogs between the application programmer and terminal user.

internationalization. (1) The process of generalizing programs or systems so that they can handle a variety of languages, character sets, and national customs. [OSF] (2) The process of designing and developing a product with a set of features, functions, and options intended to facilitate the adaptation of the product to satisfy a variety of cultural environments. [POSIX.0]

interrupt. (1) A suspension of a process, such as execution of a computer program caused by an external event, and performed in such a way that the process can be resumed. [A] (2) A signal sent by an I/O device to the processor when an error has occurred or when assistance is needed to complete I/O. An interrupt usually suspends execution of the currently executing program. [OSF] (3) In data communication, to take an action at a receiving station that causes the sending station to end a transmission. (4) To temporarily stop a process. (5) See also *signal*.

ISPF. Interactive System Productivity Facility.

ISPF/PDF. Interactive System Productivity Facility/Program Development Facility.

J

JES. Job entry subsystem.

JES2. An MVS subsystem that receives jobs into the system, converts them to internal format, selects them for execution, processes their output, and purges them from the system. In an installation with more than one processor, each JES2 processor independently controls its job input, scheduling, and output processing. See also *JES3*.

JES3. An MVS subsystem that receives jobs into the system, converts them to internal format, selects them for execution, processes their output, and purges them from the system. In complexes that have several loosely coupled processing units, the JES3 program manages processors so that the global processor exercises

centralized control over the local processors and distributes jobs to them via a common job queue. See also *JES2*.

job control. (1) Facilities for monitoring and accessing background processes. [OSF] (2) A facility that allows users to selectively stop (suspend) the execution of processes and continue (resume) their execution at a later point. The user typically employs this facility via the interactive interface jointly supplied by the terminal I/O driver and a command interpreter. Conforming implementations may optionally support job control facilities; the presence of this option is indicated to the application at compile time or run time by the definition of the [_POSIX_JOB_CONTROL] symbol. [POSIX.1]

job control language (JCL). A control language used to identify a job to an operating system and to describe the job's requirements.

job entry subsystem (JES). A system facility for spooling, job queueing, and managing I/O.

justify. (1) To control the printing positions of characters on a page so that both the left-hand and right-hand margins of the printing are regular. [I][A] (2) To shift the contents of a register or field so that the significant character at the specified end of the data is at the specified position. [T] (3) To align characters horizontally or vertically to fit the positioning constraints of a required format. [A] (4) To print a document with even right margins, even left margins, or both. (5) See *left-justify, right-justify.*

K

kernel. (1) The part of the OS/390 component containing programs for such tasks as I/O, management, and communication. The equivalent in MVS is the base control program (BCP) and Data Facility Product (DFP). (2) The part of the system that is an interface with the hardware and provides services for other system layers such as system calls, file system support, and device drivers. (3) The part of an operating system that performs basic functions such as allocating hardware resources. (4) A program that can run under different operating system environments. See also *shell*. (5) A part of a program that must be in central storage in order to load other parts of the program. (6) Synonym for *kernel address space*.

kernel address space. The address space containing the MVS support for the OS/390 UNIX services. This address space can also be called the kernel. See also *kernel*.

kernel mode. In a multiprocessor environment, the master in a master-slave relationship. The master processor operates in kernel mode, and the slave processor operates only in user mode. Kernel mode handles the interrupts and callable services. User mode informs the master when issuing a callable service. Contrast with *user mode*.

keyword. (1) A name or symbol that identifies a parameter. (2) A part of a processing statement or command operand that consists of a specific character string. (3) A part of a command operand that consists of a specific character string (such as DSNAME=). (4) A predefined word in a programming language. [OSF] (5) A reserved word. [OSF] (6) In programming languages, a lexical unit that characterizes some language construct. A keyword normally has the form of an identifier. [I]

kill. An operating system command that stops a process. [OSF]

KornShell. A command interpreter developed on UNIX, which forms the basis for the OS/390 shell.

L

left-justify. (1) To shift the contents of a register or field so that the character at the left-hand end of the data is at a specific position. [T] (2) To control the printing positions of characters on a page so that the left-hand margin of the printing is regular. [I][A]

library. (1) A collection of functions, calls, subroutines, or other data. (2) A named area on disk that can contain programs and related information (not files). A library consists of different sections, called library members. (3) A data file that contains copies of a number of individual files and control information that allows them to be accessed individually. (4) A collection of related files. For example, one line of an invoice may form an item, a complete invoice may form a file, the collection of inventory control files may form a library, and the libraries used by an organization are known as its data bank. (5) A repository for demountable recorded media, such as magnetic disk packs and magnetic tapes. [A] (6) The set of publications for a product.

line. (1) A string of characters accepted by a system as a single block of input from a terminal, such as all characters entered before a carriage return. (2) A sequence of text consisting of zero or more non-<newline> characters plus a terminating <newline> character. [POSIX.2] (3) In terminal-oriented programs, a stream of bytes terminated by <newline>. (4) A horizontal display on a screen. [OSF]

line editor. An editor that displays data one line at a time and that allows data to be accessed and modified only by entering commands. [OSF]

line mode. Synonym for *canonical mode*.

link. (1) In the file system, a connection between an inode and one or more file names associated with it. (2) Synonym for *directory entry.* (3) In data communication,

a transmission medium and data link control component that together transmit data between adjacent nodes. [OSF] (4) In programming, the part of a program that passes control and parameters between separate portions of the computer program. (5) To interconnect items of data or portions of one or more computer programs: for example, the linking of object programs by a linkage editor or linking data items by pointers. [T] (6) In SNA, the combination of the link connection and the link stations joining network nodes—for example, a System/370 channel and its associated protocols in a serial-by-bit connection under the control of Synchronous Data Link Control (SDLC).

linkage editor. (1) A computer program for creating load modules from one or more object modules or load modules by resolving cross-references among the modules and, if necessary, adjusting addresses. [T] (2) A program that resolves cross-references between separately compiled object modules and then assigns final addresses to create a single relocatable load module. If a single object module is linked, the linkage editor simply makes it relocatable. [OSF] (3) Synonymous with binder.

link-edit. To create a loadable computer program by means of a linkage editor.

link pack area (LPA). (1) An area of main storage containing reenterable routines from system libraries. Their presence in main storage saves loading time. (2) An area of virtual storage that contains reenterable routines that are loaded at IPL time and can be used concurrently by all tasks in the system.

list. (1) A sequence of one or more pipelines. (2) A data object consisting of a collection of related records.

literal. (1) A symbol or a quantity in a source program that is itself data, rather than a reference to data. (2) In programming languages, a unit that directly represents a value. For example, 14 represents the integer 14.

load module. (1) A computer program in a form suitable for loading into central storage for execution. [T] (2) A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading—for example, the input to or output from an assembler, compiler, linkage editor, or execution routine. (3) The output of the linkage editor.

local. (1) Pertaining to a device, file, or system that is accessed directly from your system, without the use of a communication line. (2) Pertaining to that which is defined and used only in one subdivision of a computer program. [T]

locale. (1) A description of a cultural environment. [POSIX.0] (2) The definition of the subset of a user's environment that depends on language and cultural conventions. [POSIX.2] (3) A tuple generally consisting of a language, territory, and codeset specification and used in internalization configuration. [OSF]

lock. A serialization mechanism by which a specific resource is restricted for the use of the holder of the lock.

logarithm. A mathematical operation related to the base of a numbering system. [OSF]

logger. (1) A functional unit that records events and physical conditions, usually with respect to time. [I][A] (2) A program that enables a user entity to log in (for example, identify itself, its purpose, and time of entry) and log out with the corresponding data. This enables the appropriate accounting procedures to be carried out in accordance with the operating system.

logical operator. A symbol that represents an operation, such as AND, OR, or NOT, on logical expressions. [OSF]

login. (1) In UNIX systems, the act of gaining access to a computer system by entering identification and authentication information at the workstation. (2) The unspecified activity by which a user gains access to the system. Each login shall be associated with exactly one login name. [POSIX.1] In the OS/390 UNIX implementation, a user gains interactive access to the shell by first logging on to a TSO/E user ID through existing documented MVS externals. Once logged on to TSO/E, a user can execute the shell by entering the OMVS command.

login name. (1) A string of characters that uniquely identifies a user to the system. (2) A user name that is associated with a login. [POSIX.1]

low-order. Least significant; rightmost. For example, in a 32-bit register (0–31), bit 31 is the low-order bit. [OSF]

M

magic number. A numeric or string constant in a file that indicates the file type.

mask. A pattern of characters that controls the keeping, deleting, or testing of portions of another pattern of characters. [I][A]

master console. In a system with multiple consoles, the basic console used for communication between the operator and the system.

member. (1) A data object in a structure, a union, or a library. (2) A partition of a partitioned data set. (3) A part of a partitioned data set that can be used independently of other members of the data set. (4) Synonym for *element*.

metacharacter. (1) A character used to specify another character or series of characters. [OSF] (2) A character that may have a special meaning in a regular expression. You can usually use a backslash to remove the special meaning.

mode. (1) A method of operation. (2) A method of operation, frequently used in UNIX to refer to read, write, run, or search permissions of a file or directory. [OSF] (3) A collection of attributes that specifies a file's type and its access permissions. [POSIX.1]

mount. (1) To make a file system accessible. [OSF] (2) To logically mount a file system in another file system with the TSO/E command MOUNT. The mount point is in a directory. (3) See also *file system, mount point*.

mountable file system. A file system stored in an HFS data set and, therefore, able to be logically mounted in another file system.

mount point. The pathname of the directory on which the file system is mounted.

MVS. Multiple Virtual Storage/Enterprise Systems Architecture.

N

named pipe. Synonym for *FIFO special file*.

Network File System (NFS**).** A protocol developed by Sun Microsystems, Inc., that allows users to directly access files on other systems in a network. [OSF] NFS provides client and server functions for distributed processing. It has transparent file sharing, lookup, and remote procedure call (RPC) interfaces.

newline. (1) A cursor-movement function that moves the cursor to the first entry field on the next line that contains an entry field. (2) The line terminator in text files and keyboard input. On the keyboard, this is generated by the <Enter> key. (3) See also *newline character*.

newline character (NL or <newline>). (1) A control character that causes the print or display position to move to the first position on the next line. This character is often represented by "\n". [OSF] (2) A character that in the output stream causes printing to start at the beginning of the next line. The <newline> is the character designated by the "\n" in the C language binding. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the next line. [POSIX.2]

NFS. Network File System.

noncanonical mode. A **tty** input processing mode where input character erase and killing are eliminated, making input characters available to the user program as they are typed. [OSF] Synonymous with character mode, raw mode. Contrast with *canonical mode*.

null character. A character with all bits set to zero [POSIX.2].

null string. (1) A string containing no element. [T] (2) The notion of a string depleted of its entities, or the notion of a string prior to establishing its entities. [A] (3) Synonymous with empty string, null character string.

null value. (1) A parameter position for which no value is specified. (2) When used with a relational data base, an indication that no data value has been assigned to the intersection of a row and a column in a relational table.

O

object code. (1) Output from a compiler or assembler which is itself executable machine code or is suitable for processing to produce executable machine code. [A] (2) Machine-executable instructions, usually generated by a compiler from source code written in a higher level language (such as C language). For programs that must be linked, object code consists of relocatable machine code. [OSF]

object module. (1) A set of instructions in machine language produced by a compiler from a source program. (2) A portion of an object program suitable as input to a linkage editor. [T]

open file. A file that is currently associated with a file descriptor. [POSIX.1]

operand. (1) An argument to a command that is generally used as an object supplying information to a utility necessary to complete its processing. Operands generally follow the options in a command line. [POSIX.2] (2) An instruction field that represents data (or the location of data) to be manipulated or operated upon. Not all instructions require an operand field. [OSF] (3) An identifier, constant, or expression that is grouped with an operator. [OSF] (4) An entity on which an operation is performed. [T][A] (5) Information entered with a command name that defines the data on which a command processor operates and that controls the execution of the command processor. (6) Information entered with a macro instruction. (7) See also *keyword, parameter*.

option. (1) A specification in a statement that can influence the execution of the statement. (2) An argument to a command that is generally used to specify changes in the *utility*'s default behavior. [POSIX.2]

OS/390 UNIX System Services. OS/390 services that support an environment within which operating systems, servers, distributed systems, and workstations share

common interfaces. OS/390 UNIX System Services supports standard application development across multivendor systems. It is required if you want to create and use applications that conform to the POSIX standard. OS/390 UNIX System Services combines the personal power of the workstation, the flexibility of open systems, and the strength of MVS. It supports and fosters a superenvironment of larger operating systems or servers and of distributed systems and workstations that share common interfaces. Users can switch back and forth between the traditional TSO/E interface and the shell interface. UNIX-skilled users can interact with the system, using a familiar set of standard commands and utilities. MVS-skilled users can interact with the system, using familiar TSO/E commands and interactive menus to create and manage hierarchical file system files and to copy data back and forth between MVS data sets and files. Application programmers and users have both sets of interfaces to choose from and, by making appropriate trade-offs, can choose to mix these interfaces.

output file. (1) A file that a program opens so that it can write to that file. (2) A file that contains the results of processing.

output redirection. The specification of an output destination other than the standard one. [OSF]

output stream. (1) Diagnostic messages and other output data issued by an operating system or a processing program on output devices especially activated for this purpose by the operator. [OSF] (2) Messages and other output data that an operating system or a processing program displays on output devices.

owner. The user who has the highest level of access authority to a data object or action, as defined by the object or action. [OSF]

P

parameter. (1) A variable that is given a constant value for a specified application and that may denote the application. [I][A] (2) A name in a procedure that is used to refer to an argument passed to that procedure. (3) An argument the user supplies to a command or function. [OSF] (4) Data passed between programs or procedures. (5) See also *operand*.

parent directory. (1) The directory one level above the current directory. (2) When discussing a given directory, the directory that both contains a directory entry for the given directory and is represented by the pathname dot-dot in the given directory. [POSIX.1] (3) When discussing other types of files, a directory containing a directory entry for the file under discussion. [POSIX.1]

parent process. A process created to carry out a program. The parent process in turn creates child processes to execute requests. See also *child process, parent process ID, process.*

parent process ID (PPID). An attribute of a new process after it is created by a currently active process. The parent process ID of a process is the process ID of its creator, for the lifetime of the creator. After the creator's lifetime has ended, the parent process ID is the process ID of an implementation-defined system process. [POSIX.1] In the OS/390 UNIX implementation, the parent process ID of the children of an ended process is set to the process ID of the INIT process, or `1`.

parse. To analyze the operands entered with a command and build a parameter list for the command processor from the information.

partitioned data set (PDS). A data set in direct access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data. Synonymous with program library.

pathname. (1) A filename specifying all directories leading to the file. (2) See also *absolute pathname, relative pathname*. (3) A filename specifying all directories leading to a file plus the filename itself. (4) A string that is used to identify a file. A pathname consists of, at most, [PATH_MAX] bytes, including the terminating null character. It has an optional beginning slash, followed by zero or more filenames separated by slashes. If the pathname refers to a directory, it may also have one or more trailing slashes. Multiple successive slashes are considered to be the same as one slash. A pathname that begins with two successive slashes may be interpreted in an implementation-defined manner, although more than two leading slashes shall be treated as a single slash. [POSIX.1] In the OS/390 UNIX implementation, the C/370 functions **fopen()**, **freopen()**, **remove()**, and **rename()** interpret names with exactly two leading slashes, no leading blanks or other characters, and the third character not a slash to mean that the rest of the name refers to a traditional MVS data set. (5) See also *absolute pathname, relative pathname*.

pattern. (1) A regular expression or series of regular expressions that define the search pattern. (2) A sequence of characters used either with regular expression notation or for pathname expansion, as a means of selecting various character strings or pathnames, respectively. The syntaxes of the two patterns are similar, but not identical; the standard always indicates the type of pattern being referred to in the immediate context of the use of the term. [POSIX.2] (3) A sequence of characters used by commands that search for strings. Some characters have special meanings in patterns; for example, `$` stands for the end of a line and `abc$` refers to the

sequence *abc* appearing at the end of a line. Some patterns can be matched by many different strings.

pattern matching. (1) The identifying of one of a predetermined set of items which has the closest resemblance to a given object, by comparing its coded representation against the representation of all the items. [T] (2) Specifying a pattern of characters that the system should find. [OSF] (3) The process of searching for strings of characters that conform to the pattern of characters in a regular expression.

PDS. Partitioned data set.

permission. (1) A code that determines how the file can be used by any users who work on the system. [OSF] (2) The modes of access to a protected object. [OSF]

PF (programmed function) key. A key on the keyboard of a display device that passes a signal to a program to call for a particular program operation.

PGID. Process group ID.

physical file. A database file that describes how data is to be presented or received from a program and how data is actually stored in the database. A physical file contains one record format and one or more members.

PID. Process ID.

pipe. (1) An interprocess communication mechanism that connects an output file descriptor to an input file descriptor. Usually the standard output of one process is connected to the standard input of another, forming a pipeline. (2) A sequence of one or more commands in FIFO order. The output of one command becomes the input to the next command. A pipe usually contains several filters. Pipes allow related or unrelated processes to read and write to each other as if they were files; they allow unidirectional communication from one process to another. OS/390 UNIX services treat pipes as though they were files. A named pipe has a directory name and is accessed by a pathname. An unnamed pipe must be used between a parent process and a child process. (3) An object accessed by one of the pair of file descriptors created by the **pipe()** function. Once a pipe is created, the file descriptors can be used to manipulate it, and it behaves identically to a FIFO special file when accessed in this way. It has no name in the file hierarchy. [POSIX.1] (4) To direct data so that the output from one process becomes the input to another process. (5) An I/O stream that has a descriptor and can be used in unidirectional communications between related processes. [OSF]

pipeline. (1) A chain of two or more processes connected by pipes. Each process in the chain acts as a filter, reading data from the standard input, performing some transformation, and writing the results to the standard output. (2) A direct, one-way connection between two or more processes. (3) A serial arrangement of processors or a serial arrangement of registers within a processor. Each processor or register performs part of a task and passes results to the next processor. Several parts of different tasks can be performed at the same time. (4) To perform processes in a series. (5) For increased processing speed, to start execution of an instruction sequence before the previous instruction sequence is completed.

pointer. (1) In the C language, a variable that holds the address of a data object or a function. (2) A physical or symbolic identifier of a unique target. (3) An identifier that indicates the location of an item of data. [A] (4) A data element that indicates the location of another data element. [T] (5) In computer graphics, a manually operated functional unit used to specify an addressable point. A pointer may be used to conduct interactive graphic operations, such as selection of one member of a predetermined set of display elements, or indication of a position on a display space while generating coordinate data. [T]

polling. Interrogation of devices for such purposes so as to avoid contention, to determine operational status, or to determine readiness to send or receive data. [A]

portability. (1) The ability to run a program on more than one computer without modifying it. (2) The ability to use applications, data sets, or files with different operating systems. (3) The ease with which software can be transferred from one information system to another. [POSIX.0]

portable character set. The set of characters described in POSIX.2 2.4 that is supported on all conforming systems. See also *portable filename character set.* [POSIX.2]

portable filename character set. The set of characters from which portable filenames are constructed. For a filename to be portable across conforming implementations of POSIX.1, it shall consist only of the uppercase and lowercase characters of the alphabet (*A* through *Z* and *a* through *z*), the digits *0* through *9,* the period (**.**), the underscore (_), and the hyphen (**-**). The hyphen shall not be used as the first character of a portable filename. Uppercase and lowercase letters shall retain their unique identities between conforming implementations. In the case of a portable pathname, the slash character may also be used. [POSIX.1]

positional parameter. (1) A shell facility that assigns values from the command line to variables in a program. [OSF] (2) A parameter that must appear in a specified location relative to other positional parameters. (3) One of the command line arguments to a shell file. Positional parameters are referenced by `$1`, `$2`, and so on.

POSIX. Portable Operating System Interface for Computer Environments, an interface standard

governed by the IEEE and based on UNIX. POSIX is not a product. Rather, it is an evolving family of standards describing a wide spectrum of operating system components ranging from C language and shell interfaces to system administration.

PPID. Parent process ID.

process. (1) A function being performed or waiting to be performed. (2) An executing function, or one waiting to execute. (3) A function, created by a **fork()** request, with three logical sections:
- Text, which is the function's instructions.
- Data, which the instructions use but do not change.
- Stack, which is a push-down, pop-up save area of the dynamic data that the function operates upon.

(4) A program using OS/390 UNIX services. The program can be created by a **fork()** function or fork callable service, or the program can be dubbed because it requested OS/390 UNIX services. The three types of processes are:
- User processes, which are associated with a user at a workstation
- Daemon processes, which do systemwide functions in user mode, such as printer spooling
- Kernel processes, which do systemwide functions in kernel mode, such as paging

A process can run in a user address space, a forked address space, or a kernel address space. In an MVS system, a process is handled like a task. See also *task*. (5) An address space and one or more threads of control that execute within that address space, and their required system resources. [POSIX.0] (6) An address space and single thread of control that executes within that address space, and its required system resources. A process is created by another process issuing the **fork()** function. The process that issues **fork()** is known as the parent process, and the new process created by the **fork()** is known as the child process. [POSIX.1] (7) A sequence of actions required to produce a desired result. [OSF] (8) An entity receiving a portion of the processor's time for executing a program. [OSF] (9) An activity within the system that is started by a command, a shell program, or another process. Any running program is a process. (10) A unique, finite course of events defined by its purpose or by its effect, achieved under given conditions. (11) Any operation or combination of operations on data. (12) The current state of a program that is running—including a memory image, the program data, the variables used, the general register values, the status of opened files used, and the current directory. Programs running in a process must be either operating system programs or user programs. [OSF] (13) A running program, including the memory occupied, the open files, the environment, and other attributes specific to a running program.

process group. A collection of processes that permits the signaling of related processes. Each process in the system is a member of a process group that is identified by a process group ID. A newly created process joins the process group of its creator. [POSIX.1]

process group ID (PGID). The unique identifier representing a process group during its lifetime. A process group ID is a positive integer that can be contained in a *pid_t*. It shall not be reused by the system until the process group lifetime ends. [POSIX.1]

process ID (PID). (1) A unique number assigned to a process that is running. [OSF] (2) The unique identifier representing a process. A process ID is a positive integer that can be contained in a *pid_t*. A process ID shall not be reused by the system until the process lifetime ends. In addition, if there exists a process group whose process group ID is equal to that process ID, the process ID shall not be reused by the system until the process group lifetime ends. A process that is not a system process shall not have a process ID of 1. [POSIX.1]

The equivalent in MVS is an address space identifier (ASID).

profile. (1) A file containing customized settings for a system or user. [OSF] (2) Data that describes the significant characteristics of a user, a group of users, or one or more computer resources. (3) A set of instructions to initialize a user's shell session. The shell automatically reads and executes these commands if this file is in the user's home directory. (4) In computer security, a description of the characteristics of an entity to which access is controlled. (5) A set of one or more base standards, and, where applicable, the identification of chosen classes, subsets, options, and parameters of those base standards, necessary for accomplishing a particular function. [POSIX.0]

prompt. (1) A displayed symbol or message that requests information or operator action. [OSF] (2) A message issued to a terminal user requesting information necessary to continue processing. (3) A common action that users request while the cursor is in an entry field. A prompt produces a menu of available choices for that entry field. Users can select a choice from the menu to insert in the entry field.

pseudoterminal (pty). A special file in the **/dev** directory that effectively functions as a keyboard and display device. Synonymous with pseudo-TTY.

pseudo-TTY. Synonym for *pseudoterminal*.

pty. Pseudoterminal.

Q

qualifier. (1) A unique name used to identify another name. [OSF] (2) A modifier that makes a name unique. (3) All names in a qualified name other than the rightmost, which is called the simple name.

quote. To mask the special meaning of certain characters, causing them to be taken literally. [OSF]

R

RACF. Resource Access Control Facility.

record. (1) In programming languages, an aggregate that consists of data objects, possibly with different attributes, that usually have identifiers attached to them. [I] (2) A set of data treated as a unit. [T] (3) A collection of fields treated as a unit. [OSF] (4) A self-contained collection of information about a single object. A record is made up of a number of distinct items, called fields. A number of shell programs (for example, **awk**, **join**, and **sort**) are designed to process data consisting of records separated by newlines, where each record contains a number of fields separated by spaces or some other character. **awk** can also handle records separated by characters other then newlines. (5) See *fixed-length record, variable-length record.* (6) Using a function to define itself. [OSF]

redirect. To divert data from a process to a file or device to which it would not normally go. [OSF]

redirection. (1) A system profile construction method of starting at a base platform and adding new services by allowing a service component to ask the base platform to redirect all requests for that type of service to the service component. [POSIX.0] (2) Changing the association between files and file descriptors for a program. A process inherits file descriptors from the process that created the program (usually the shell). A file descriptor's standard input and standard output are usually associated with the keyboard and display screen, respectively. The shell can arrange for these descriptors (or any others) to be associated with other files before creating the new process. In particular, `<infile` redirects the standard input from **infile**, while `>outfile` redirects the standard output to **outfile**.

regular expression. (1) A pattern (sequence of characters or symbols) constructed according to the rules defined in POSIX.2 2.8. [POSIX.2] (2) A set of characters, metacharacters, and operators that define a string or group of strings in a search pattern. [OSF] (3) A string containing wildcard characters and operations that define a set of one or more possible strings. [OSF] (4) A more technical term for *pattern*. (5) See also *wildcard character*.

regular file. A file that is a randomly accessible sequence of bytes, with no further structure imposed by the system. [POSIX.1]

relative pathname. (1) The name of a directory or file expressed as a sequence of directories followed by a filename, beginning from the current directory. Relative pathnames do not begin with a / (slash) but are relative to the current directory. (2) A pathname that does not begin with a slash. The predecessor of the first filename in the pathname is taken to be the current working directory of the process. [POSIX.1]

Resource Access Control Facility (RACF). An IBM-licensed program that provides for access control by identifying and by verifying the users to the system, authorizing access to protected resources, and logging the detected unauthorized access to protected resources.

right-justify. (1) To control the positions of characters on a page so that the right-hand margin of the printing is regular. [I][A] (2) To shift the contents of a register or field so that the character at the right-hand end of the data is at a specific position. [T] (3) To align characters horizontally so that the rightmost character of a string is in a specified position. [A]

root. (1) The starting point of the file system. (2) The first directory in the system. (3) The user name for the system user with the most authority.

root directory. (1) The directory that contains all other directories in the system. The root directory in a hierarchical file system is analogous to the MVS master catalog. (2) The lowest directory in the file system hierarchy. It is referred to as "/". (3) A directory, associated with a process, that is used in pathname resolution for pathnames that begin with a slash. [POSIX.1] (4) The top directory in the file system tree. UNIX and POSIX-conforming systems have a single root directory, with mounted devices. (5) See also *effective root directory*.

root file system. The basic file system, onto which all other file systems can be mounted. The root file system contains the operating system files that get the rest of the system running.

row. A horizontal arrangement of characters or other expressions. [A] Contrast with *column.*

S

S_ISGID bit. Synonym for *set-group-ID mode bit.*

S_ISUID bit. Synonym for *set-user-ID mode bit.*

scroll. (1) To move a display image vertically or horizontally to view data that otherwise cannot be observed within the boundaries of the display screen. (2) To move the representation of data vertically or horizontally relative to the terminal screen. There are two types of scrolling: (a) the cursor moves with the data; or (b) the cursor remains stationary while the data moves. [POSIX.2]

search path. The sequence of directories that a command interpreter should search to find the program that the user wants to run.

security administrator. A programmer who manages, protects, and controls access to sensitive information.

separator. (1) A punctuation character that separates parts of a command or file. (2) A punctuation character used to delimit character strings.

sequential data set. (1) A data set whose records are organized on the basis of their successive physical positions, such as on magnetic tape. Contrast with *direct data set*. (2) A data set in which the contents are arranged in successive physical order and are stored as an entity. The data set can contain data, text, a program, or part of a program. Contrast with *partitioned data set (PDS).*

session. (1) The period of time during which a user of a terminal can communicate with an interactive system—usually, the elapsed time between logon and logoff. (2) The period of time during which programs or devices can communicate with each other. [OSF] (3) A collection of process groups established for job control purposes. Each process group is a member of a session. Each process is considered to be a member of the session of which its process group is a member. A newly created process joins the session of its creator. A process can alter its session membership. Implementations that support the **setpgid()** function can have multiple process groups in the same session. [POSIX.1] Every process group, and associated process, belongs to a session. Any new process also belongs to the session of the process that created it. (4) In network architecture, an association of facilities that establish, maintain, and release connections for communication between stations. [OSF] (5) In SNA, a logical connection established between two network addressable units (NAUs) that allows them to communicate. For routing purposes each session is identified by the local or network addresses of the session partners.

setgid bit. Deprecated term for *set-group-ID mode bit*.

set-group-ID mode bit. In setting file access permissions, the bit that sets the effective group ID of the process to the file's group on execution. Synonymous with S_ISGID bit.

setuid bit. Deprecated term for *set-user-ID mode bit*.

set-user-ID mode bit. In setting file access permissions, the bit that sets the effective user ID of the process to the file's owner on execution. Synonymous with S_ISUID bit.

sh_history. See *history file*.

shell. (1) A program that interprets and processes interactive commands from a pseudoterminal or from lines in a shell script. The equivalent in MVS is Time Sharing Option (TSO) and Interactive System Productivity Facility (ISPF). (2) A program that interprets sequences of text input as commands. It may operate on an input stream, or it may interactively prompt and read commands from a terminal. [POSIX.2] Synonymous with command language interpreter. (3) A software interface between a user and the operating system of a computer. Shell programs interpret commands and user interactions on devices such as keyboards, pointing devices and touch-sensitive screens and communicate them to the operating system. (4) The command interpreter that provides a user interface to the operating system and its commands. (5) The program that reads a user's commands and executes them. (6) The shell command language interpreter, a specific instance of a shell. [POSIX.2] (7) A layer, above the kernel, that provides a flexible interface between users and the rest of the system. (8) Software that allows a kernel program to run under different operating system environments. (9) See also *interface, shell program, KornShell*.

shell program. A program that accepts and interprets commands for the operating system. See also *shell*.

shell prompt. The character string on the command line indicating that the system can accept a command.

shell script. (1) A file of shell commands. If the file is executable, a user can run it by specifying the file's name as a shell command or as an operand on **sh** or on the TSO/E OMVS command. A shell script is like a TSO/E REXX program. (2) A file containing shell commands. If the file is made executable, it can be executed by specifying its name as a simple command: Execution of a shell script causes a shell to execute the commands within the script. Alternately, a shell can be requested to execute the commands in a shell script by specifying the name of the shell script as the operand to the **sh** utility. [POSIX.2]

shell variables. Facilities of the shell program for assigning variable values to constant names.

signal. (1) A means of informing processes of asynchronous events. (2) A mechanism by which a process may be notified of, or affected by, an event occurring in the system. Examples of such events include hardware exceptions and specific actions by processes. The term *signal* is also used to refer to the event itself. [POSIX.1] (3) An indication that an asynchronous event completed. A signal is sent to a process. Signals are simulations of *interrupts*. (4) A simple method of communication between two processes. One process can inform the other process when an event occurs. [OSF] (5) A method of interprocess communication that simulates software interrupts. Contrast with *interrupt*.

single-quote. The character ' —also known as *apostrophe*. [POSIX.2]

slash. (1) The literal character "/". This character is also known as *solidus* in ISO 8859-1 [B34] [POSIX.1]

(2) The character /. UNIX and POSIX-conforming systems use the slash to separate the parts of a filename.

socket. (1) A method of communication between two processes. Sockets allow communication in two directions, in contrast to pipes, which allow communication in only one direction. The processes using a socket can be on the same system or on systems in the same network. (2) A unique host identifier created by the concatenation of a port identifier with a TCP/IP address. (3) An interface, linked with TCP/IP or other protocols, that allows processes on different machines to communicate. Sockets are similar to APPC, but the communication mechanism is transparent: It consists of three layers (socket layer, protocol layer, and device layer). There are two types of sockets: virtual streak sockets and datagram sockets. (4) A port on a specific host; a communication endpoint that is accessible through a protocol family's addressing mechanism. (5) In TCP/IP, the Internet address of the host computer on which the application runs, and the port number it uses. (6) In interprocess communication, an endpoint of communication. (7) The system call that creates a socket and its associated data structure. (8) A port identifier. (9) Synonym for *port*.

spawn. To create and start a child process running a named program. The **spawn()** function is the logical combination of fork and exec; its purpose is to avoid the system overhead incurred with fork. The MVS equivalent is multitasking, or attaching.

special character. A character other than a letter or number. For example, *, +, and % are special characters.

standard error (stderr). (1) The place where many programs place error messages: the display screen unless another place is specified with redirection. [OSF] (2) An output stream usually intended to be used for diagnostic messages. [POSIX.2] (3) The conventional name for file descriptor 2. By convention, programs write diagnostics and error messages to this descriptor. Usually, the descriptor refers to the display screen, but it may be changed by redirection. This descriptor is separate from standard output so that error diagnostics are still visible when the output is redirected.

standard input (stdin). (1) The primary source of data going into a command. Standard input comes from the keyboard unless redirection or piping is used, in which case standard input can be from a file or the output from another command. (2) An input stream usually intended to be used for primary data input. [POSIX.2] (3) The conventional name for file descriptor 0. By convention, programs read input from this descriptor. Usually, the descriptor refers to the keyboard, but it may be changed by redirection.

standard output (stdout). (1) The primary destination of data coming from a command. Standard output goes to the display unless redirection or piping is used, in which case standard output can be to a file or another command. (2) An output stream usually intended to be used for primary data output. [POSIX.2] (3) The conventional name for file descriptor 1. By convention, programs write output to this descriptor. Usually, the descriptor refers to the display screen, but may be changed by redirection.

stderr. Standard error.

stdin. Standard input.

stdout. Standard output.

STEPLIB. A set of private libraries used to store a new or test version of an application program, such as a new version of a runtime library.

STEPLIB environment. The STEPLIB DD and associated private library allocations that make up a user's MVS program search order environment.

sticky bit. A file access permission bit that allows multiple users to share a single copy of an executable file. Only someone with root authority can set the sticky bit.

stop. To end, in a controlled manner, the current processing activity in a computer system because it is impossible or undesirable for the activity to proceed. [OSF] Synonymous with abort.

stream. (1) An ordered sequence of characters, as described by the C Standard. [POSIX.2] (2) Sequential input or output from an open file descriptor. [OSF] (3) All data transmitted through a data channel in a single read or write operation. [OSF] Synonymous with data stream. (4) A full-duplex connection between a user process and device or pseudodevice. A stream consists of several linearly connected modules, and is analogous to a shell pipeline, except that data flows in both directions. The modules communicate almost exclusively by passing messages to their neighbors. [OSF]

stream editor. An editor invoked by the **sed** command, which modifies lines from a specified file, according to an edit script, and writes them to a standard output. [OSF]

string. (1) A linear sequence of entities such as characters or physical elements. Examples of strings are alphabetic string, binary element string, bit string, character string, search string, and symbol string. [OSF] (2) An ordered sequence of bits, octets, or characters, accompanied by the string's length. [OSF] (3) A sequence of characters or numbers. Any sequence of characters, as in `abc`. For the shell, strings should be enclosed by quotes (") to hide any blanks or tabs in the string from the shell.

structure. (1) A variable that contains an ordered group of data objects. Unlike an array, the data objects within a structure can have varied data types. (2) Synonym for *data structure*.

subcommand. A request for an operation that is within the scope of work requested by a previously issued command.

subdirectory. In the file system hierarchy, a directory contained within another directory. Directories may be nested to arbitrary depth.

subshell. An instance of the shell program started from an existing shell program.

substring. A part of a character string.

suffix. A character string attached to the end of a filename that helps identify its file type. [OSF] For example, in **/dir/file.c**, the suffix is **c**.

superuser. (1) A system user who operates without restrictions. A superuser has the special rights and privileges needed to perform administrative tasks. The MVS equivalent is a user in privileged, or supervisor, mode. (2) A system user who can pass all OS/390 UNIX security checks. A superuser has the special rights and privileges needed to manage processes and files.

switch. To change processing from the OS/390 shell to a TSO/E command mode or to subcommand mode, or to change back to the shell from one of those modes. In TSO/E command mode, you can enter TSO/E commands; in subcommand mode, you can enter OMVS subcommands.

symbolic link. A type of file system entry that contains the pathname of and acts as a pointer to another file or directory. [OSF]

synchronous transmission. (1) In data communication, a method of transmission in which the sending and receiving of characters is controlled by timing signals. A predetermined number of bits is sent across the line per second, for example, 2400 bits per second. There are no start and stop bits. Contrast with *asynchronous transmission*. (2) Data transmission in which the time of occurrence of each signal representing a bit is related to a fixed time base. [I]

syscall. Synonym for *callable service*.

system call. Synonym for *callable service*.

T

table. (1) An array of data each item of which can be unambiguously identified by means of one or more arguments. [I][A] (2) A two-dimensional array in which each item and its position with respect to other items is identified.

text file. A file that contains characters organized into one or more lines. The lines do not contain NUL characters and none shall exceed {LINE_MAX} bytes in length, including the <newline>. Although POSIX.1 does not distinguish between text files and binary files, many utilities produce predictable or meaningful output only when operating on text files. The standard utilities that have such restrictions always specify *text files* in their Standard Input or Input Files subclauses. [POSIX.2] See also *binary file*.

thread. A stream of computer instructions that is in control of a process. A multithreaded process begins with one stream of instructions (one thread) and may later create other instruction streams to perform tasks.

TTY. Any device that uses the **termios** standard terminal device interface. TTY devices typically perform input and output on a character-by-character basis.

tilde. The character ~ [POSIX.2]

Time Sharing Option (TSO). An operating system option; for the System/370* system, the option provides interactive time sharing from remote terminals.

Time Sharing Option Extensions (TSO/E). (1) The base for all TSO enhancements. It provides MVS users with additional functions, improved usability, and better performance. (2) In the MVS environment, TSO/E also provides virtual storage constraint relief.

Transmission Control Protocol (TCP). A communications protocol used in Internet and any other network following the U.S. Department of Defense standards for internetwork protocol. TCP provides a reliable host-to-host protocol in packet-switched communication networks and in an interconnected system of such networks. It assumes that the Internet Protocol is the underlying protocol. The protocol that provides a reliable, full-duplex, connection-oriented service for applications.

Transmission Control Protocol/Internet Protocol (TCP/IP). The two fundamental protocols of the Internet protocol suite. The abbreviation TCP/IP is frequently used to refer to this protocol suite. TCP provides for the reliable transfer of data, while IP transmits the data through the network in the form of datagrams. Users can send mail, transfer files across the network, or execute commands on other systems. [OSF]

TSO/E. Time Sharing Option Extensions.

TTY. Any device that uses the **termios** standard terminal device interface. TTY devices typically perform input and output on a character-by-character basis.

U

UID. See *user ID*.

unformatted file. A file displayed with the data that is not arranged with particular characters. Contrast with *formatted file*.

unmount. To logically disassociate a mountable file system from another file system. The TSO/E command to perform this action is UNMOUNT or UMOUNT.

user address space. An address space that has at least one MVS task known to the kernel. This address space can contain a shell or an application program that uses kernel services.

user ID. (1) A unique string of characters that identifies an operator to the system. This string of characters limits the functions and information the operator can use. (2) A nonnegative integer, which can be contained in an object of type *uid_t,* that is used to identify a system user. When the identity of the user is associated with a process, a user ID value is referred to as a real user ID, an effective user ID, or an (optional) saved set-user-ID. [POSIX.1] (3) The identification associated with a user or job. The two types of user IDs are:

- **TSO/E user ID:** A string of characters that uniquely identifies a TSO/E user or a batch job owner to the security program for the system. The batch job owner is specified on the USER parameter on the JOB statement or inherited from the submitter of the job. This user ID identifies a RACF user profile.
- **OS/390 UNIX ID:** A fullword integer that the security administrator assigns to each MVS user ID. This integer, referred to as the UID, is the sole basis for authority checking against such POSIX-defined resources as hierarchical files.

A user ID is equivalent to an account on a UNIX-type system. (4) A symbol identifying a system user. (5) Synonymous with user identification.

user mode. In a multiprocessor environment, the slave in a master-slave relationship. One processor operates in kernel mode, the other (slave) in user mode. Kernel mode handles the interrupts and callable service. User mode informs the master when making a callable service. In user mode, a process is carried out in the user's program rather than in the kernel. Contrast with *kernel mode*.

user name. (1) The 1-to-8-character name of the owner of an user address space. The OS/390 UNIX user name is:

- For an OS/390 UNIX application running in batch, the user ID in the USER parameter on the JOB statement or the TSO/E user ID for an interactive submitter of the job.
- For an interactive application, the OS/390 UNIX user name is the TSO/E user ID for the user.

(2) In RACF, one to twenty alphanumeric characters that represent a RACF-defined user. (3) A string that is used to identify a user. [POSIX.1] (4) Synonym for *user ID*.

user profile. (1) A description of a user, including user ID, user name, default group name, password, owner, access authority, and other attributes obtained at logon. (2) A file in the user's home directory named **.profile** that contains shell commands that set initial user-defined characteristics and defaults for the session.

V

variable-length record. (1) A record having a length independent of the length of other records with which it is logically or physically associated. (2) Pertaining to a file in which the records need not be uniform in length. [A] (3) Contrast with *fixed-length record.*

vertical tab. The vertical tab character (<vertical tab>). [POSIX.2]

W

wait. (1) A state allowing a parent process to synchronize with the execution of an exit issued by a child process. (2) A state in which no instructions are fetched or executed. Synonymous with wait state.

white space. (1) Space characters, tab characters, newline characters, and comments. [OSF] (2) A sequence of one or more spaces or horizontal tab characters. White space is used to separate commands on the command line. (3) A sequence of one or more characters that belong to the **space** character class as defined via the LC_CTYPE category in the current locale. In the POSIX Locale, white space consists of one or more <blank>s (<space>s and <tab>s), <newline>s, <carriage return>s, <form-feed>s, and <vertical tab>s. [POSIX.2]

wildcard character. Either the * or ? special character that can be used in a file specification to match one or more characters. For example, placing ? in a file specification means any character can be in that position. For another example, the file selector `?a` matches the filenames **aa**, **Ba**, **ca**, and so on. or more characters; thus the file selector `a*` matches any name beginning with **a**. Synonymous with global character (in an MVS context), pattern-making character. See also *regular expression*.

working directory. (1) The active directory used to resolve pathnames that do not begin with a slash. In similar systems, a working directory may be called the *current directory* or the *current working directory*. (2) A directory, associated with a process, that is used in pathname resolution for pathnames that do not begin with a slash. [POSIX.1] (3) Synonym for *current directory*. (4) See also *current directory*.

write access. In computer security, permission to write to an object. Synonymous with write permission.

write permission. Synonym for *write access*.

Numerics

3270 passthrough mode. A mode that lets a program running from the OS/390 shell send and receive a **3270** data stream or issue TSO/E commands.

Index

Special Characters

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Readers' Comments — We'd Like to Hear from You

OS/390
UNIX System Services
User's Guide

Publication No. SC28-1891-09

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name _____ Address _____

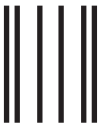
Company or Organization _____

Phone No. _____

Readers' Comments — We'd Like to Hear from You
SC28-1891-09

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape



NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Department 55JA, Mail Station P384
2455 South Road
Poughkeepsie, NY
12601-5400

Fold and Tape | **Please do not staple** | Fold and Tape

Cut or Fold
Along Line

Program Number: 5647-A01

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC28-1891-09