z/VM

OpenExtensions User's Guide

version 6 release 1

SC24-6206-00

z/VM

OpenExtensions User's Guide

version 6 release 1

SC24-6206-00

Note:
Before using this information and the product it supports, read the information in “Notices” on page 205.

Acknowledgments

InterOpen/POSIX Shell and Utilities is a source code product providing POSIX.2 (Shell and Utilities) functions to the OpenExtensions services offered with z/VM. InterOpen/POSIX Shell and Utilities is developed and licensed by Mortice Kern Systems (MKS) Inc. of Waterloo, Ontario, Canada.

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.

This edition applies to version 6, release 1, modification 0 of IBM z/VM (product number 5741-A07) and to all subsequent releases and modifications until otherwise indicated in new editions.

This edition replaces SC24-6108-01.

Contents

Figures

Tables

About this Document

This document provides information for setting up the IBM® z/VM® OpenExtensions™ facilities and using the OpenExtensions shell. This information helps users use 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, other support services associated with OpenExtensions are also described.

This document describes how to use the OpenExtensions shell, the file system, and communication services. Using this information, 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/C++ language program.
- Run shell scripts and C/C++ language programs interactively (in the foreground), in the background, or in batch.
- Switch easily between the shell and CMS.
- Move CMS record file into the file system, or move files from the byte file system (BFS) into CMS.
- Enter shell commands or CMS commands from the shell command line.
- Use XEDIT to create or to edit a file in the file system.
- Manage your file system.

Intended Audience

Information on setting up OpenExtensions facilities is provided for systems programmers and system administrators. Information on using the OpenExtensions shell is provided for application programmers and end users.

Conventions Used in This Document

The following conventions are used in this document.

Escape Character Notation

When you see the following notation:

enter <EscChar-C>

it should be interpreted as:

type the EscChar, which by default is the **¢** (cent sign) and then type the **C** character.

Press ENTER after typing these characters.

Note: To change the escape character to something other than the cent sign, see the BPX1TSX service in *z/VM: OpenExtensions Callable Services Reference*.

Case-Sensitivity

The OpenExtensions shell commands and CMS OPENVM commands are case-sensitive and distinguish characters as either uppercase or lowercase. Therefore, `FILE1` is not the same as `file1`.

Typography

The following typographic conventions are used:

BOLD	Bold and uppercase is used for all command names (**OPENVM SHELL**) except the shell commands, statements (**CLINKNAME**), and references to a key that you would press (**ENTER**).
bold	Bold and lowercase is used for shell commands (**make**).
variable	Lowercase italics is used to indicate a variable.
VARIABLE	Uppercase italics is used to indicate a shell environment variable.
`example font`	Example font is used to indicate file specifications (`.profile`, `XEDIT PROFILE`), directory names (`/usr/lib/nls/charmap`), and verbatim user input.

Where to Find More Information

For detailed reference information on OpenExtensions shell commands and utilities and CMS OPENVM commands, see the *z/VM: OpenExtensions Commands Reference*.

For more extensive information on using the **lex**, **yacc**, and **make** utilities, see *z/VM: OpenExtensions Advanced Application Programming Tools*.

For a list of other z/VM publications, see “Bibliography” on page 211.

Links to Other Online Documents

If you are viewing the Adobe® Portable Document Format (PDF) version of this document, it might contain links to other documents. A link to another document is based on the name of the requested PDF file. The name of the PDF file for an IBM document is unique and identifies the edition. The links provided in this document are for the editions (PDF names) that were current when the PDF file for this document was generated. However, newer editions of some documents (with different PDF names) might exist. A link from this document to another document works only when both documents reside in the same directory.

How to Send Your Comments to IBM

We appreciate your input on this publication. Feel free to comment on the clarity, accuracy, and completeness of the information or give us any other feedback that you might have.

Use one of the following methods to send us your comments:

1. Send an e-mail to mhvrcfs@us.ibm.com
2. Visit the z/VM reader's comments Web page at www.ibm.com/systems/z/os/zvm/zvmforms/webqs.html
3. Mail the comments to the following address:
   IBM Corporation
   Attention: MHVRCFS Reader Comments
   Department H6MA, Mail Station P181
   2455 South Road
   Poughkeepsie, NY 12601-5400
   U.S.A.
4. Fax the comments to us as follows:
   From the United States and Canada: 1+845+432-9405
   From all other countries: Your international access code +1+845+432-9405

Include the following information:

- Your name and address
- Your e-mail address
- Your telephone or fax number
- The publication title and order number:
  z/VM V6R1 OpenExtensions User's Guide
  SC24-6206-00
- The topic and page number related to your comment
- The text of your comment

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.

IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you submit to IBM.

If You Have a Technical Problem

Do not use the feedback methods listed above. Instead, do one of the following:

- Contact your IBM service representative.
- Contact IBM technical support.
- Visit the z/VM support Web page at www.vm.ibm.com/service/
- Visit the IBM mainframes support Web page at www.ibm.com/systems/support/z/

Part 1. Setting Up OpenExtensions

Chapter 1. Setting Up OpenExtensions

This chapter describes the tasks involved in setting up the OpenExtensions facilities in z/VM that allow users to run POSIX applications. These tasks involve assigning POSIX security values to users and setting up the OpenExtensions Byte File System (BFS). Any CMS user can run a POSIX application, but to assure proper access to resources and to allow the management of POSIX data, the tasks described in this chapter must be performed. These tasks involve specifying certain system configuration file statements and CP directory control statements that are described in *z/VM: CP Planning and Administration*.

Understanding the POSIX Database Concepts

Two important tasks involved in enabling an installation to make full use of POSIX are:

- Setting up the POSIX user database
- Setting up the POSIX group database

If an External Security Manager (ESM) is installed, it may be capable of managing these databases and providing CP with information from them upon request. If not, then CP will obtain this information from the CP user directory.

POSIX User Database

The user database contains POSIX information about the users in the system. For each user, it contains at least the following information:

User name
: This is the login name that identifies a POSIX user. It is analogous to a user's user ID on a VM system and is the lowercase version of the VM user ID.

User ID (UID)
: This is a numeric identifier for the POSIX user. It will be the user's initial real UID, effective UID and saved set-UID when the user logs on. It identifies the user to the system when certain POSIX functions are being handled, including authorization checks before file access and program execution. Even though multiple users are permitted to have the same UID, this is **not recommended**, because UIDs are used for various authorizations. If multiple users have the same UID, individual accountability may be lost.

  A UID of 0 identifies a user as one with "appropriate privileges". POSIX permits processes with appropriate privileges to perform additional or different functions in certain situations. Take care when assigning a UID of 0 to a user. It may be appropriate for certain service virtual machines.

  If users are not assigned a UID in the user database, they may each be assigned the same default value by the system. This is what happens when the user database information is contained in the CP directory. If an ESM provides this information, it may behave differently. If multiple users have the same UID, each of them will appear the same to POSIX functions that reference these UIDs. As previously mentioned, this is not recommended.

Primary group id (GID)
: This identifies a POSIX group defined in the POSIX group database. It will be the user's initial real GID, effective GID and saved set-GID when the

user logs on. A user's group affiliation is referenced when certain POSIX functions are being handled, including authorization checks before file access and program execution.

If users are not assigned a GID in the user database, they may each be assigned the same default value by the system. In this case, all of these users will appear the same to POSIX functions that reference these GIDs. This could be used to permit all these users to access or run certain files, or it could be used to deny all "unregistered" users from accessing or running any POSIX file.

Initial working directory
This is the user's home directory. Unless overridden by the OPENVM SET DIRECTORY command, it will be the current directory when a user first enters the POSIX environment.

Initial user program
This is the name of an application. It is typically a shell, a program that accepts commands from the user and supervises the execution of other programs. This is the program that will be invoked by the OPENVM SHELL command.

In addition to the above information required by POSIX, the following information may also be contained in the user database:

File system root
This is the user's root file system. Unless overridden by the OPENVM MOUNT command, it is the Byte File System that will be mounted as the root file system when a user first enters the POSIX environment.

When not provided by an ESM, most of the user database information is taken from POSIXINFO directory control statements in the individual users' directory entries. Because some of the database information may consist of very long, mixed-case character strings with embedded blanks and quotation marks, it may be necessary to specify multiple POSIXINFO statements or continue a single statement across multiple records in the user directory file.

POSIX Group Database

The group database defines the POSIX groups that exist on the system. For each group, it contains at least the following information:

Group name
This is the name of the POSIX group.

Group ID (GID)
This is a numeric identifier for the POSIX group. Multiple groups with the same GID are permitted to be defined. Care should be taken when doing this, because it may lead to unexpected results during certain POSIX functions.

Member list
This is a list of all the users who are members of the POSIX group.

When not provided by an ESM, the group database information is taken from POSIXGROUP and POSIXGLIST directory control statements.

POSIX includes the concept of supplementary groups. Some of the groups of which a user is a member are defined to be that user's supplementary groups. This list of groups is referenced when certain POSIX functions are being handled, including

authorization checks before file access and program execution. When not provided by an ESM, this list is derived from the user's POSIXGLIST and POSIXINFO directory control statements.

POSIX Set and Query Functions

Certain set and query functions exist for the database and POSIX process information. Controls are provided to allow an installation to permit or prohibit these on a system-wide or individual user ID level. The USER_DEFAULTS POSIXOPT statement in the system configuration file can be used to specify the system default authorization for certain POSIX query and set functions. The POSIXOPT directory control statement can be used to specify these POSIX authorizations for a single user.

This support may require the use of continued directory control statements and mixed-case operands on directory control statements. Some directory statements are permitted to be continued across multiple records in the source directory file. Certain operands on some statements are case-sensitive, and their case is preserved by the DIRECTXA utility and the CP functions that return them to a guest program. Due to these characteristics, care must be taken when editing a directory file containing these statements and operands.

DIRPOSIX Utility

The DIRPOSIX utility is provided to aid the system programmer in the assigning of UIDs and GIDs to the users. It will aid in the migration to a POSIX environment. Use DIRPOSIX to add POSIX information to a user directory source file. It performs the following functions:

- Assigns a unique UID to each user ID that has no UID specification and is not listed in the DIRPOSIX USEREXCL file
- Assigns a primary group to each user ID that has no primary group specification and is not listed in the DIRPOSIX USEREXCL file
- Adds the standard "system" group definitions, if they do not already exist
- Adds the standard "system" user definitions, if they do not already exist

DIRPOSIX provides a mechanism for reserving installation-specified UIDs. It will not assign any UIDs listed in the DIRPOSIX UIDEXCL file.

For more information, see Appendix A, "DIRPOSIX Utility," on page 159.

Assigning POSIX User IDs to VM Users

This task involves assigning a UID to each user who will be using POSIX applications. While each user's UID need not be unique, you should make each unique unless you have a specific reason to do otherwise. Users with the same UID will be seen as having the same access to files in BFS. The only UID with special meaning is the UID of 0, which denotes a superuser. Such a user can access any file in BFS and perform other restricted functions. Only trusted users should be assigned a UID of 0.

If an ESM is installed to maintain the POSIX user database, refer to the ESM documentation for instructions on assigning UIDs. Otherwise, the UIDs are assigned in user entries in the CP directory. Any user not explicitly assigned a UID will automatically be assigned the default UID of 4294967295 (X'FFFFFFFF').

Defining POSIX User Groups

This task involves assigning a GID to a group name. If an ESM is not installed to handle the POSIX group database, this is done in the global definition section of the CP directory with a POSIXGROUP statement. Duplicate group names are not permitted, but multiple groups may have the same GID. Groups that have the same GID are considered to be the same when performing file access permission checking. However, they are treated as different groups during database queries such as `getgrnam()` and `getgrgid()`. In addition, certain queries that require a GID as input may return information about a group other than the intended one. For these reasons, you must take care when assigning the same GID to more than one group.

Assigning VM Users to POSIX User Groups

The assignment of users to user groups should be based on common access to data. The way to share BFS files with other users is to give selected file access permissions to the members of the file's user group. User groups define the set of users that have common file use needs.

If an ESM is installed to maintain the POSIX group database, refer to the ESM documentation for information on how to define groups. Otherwise, a user's primary group is assigned by specifying a group name or GID on a POSIXINFO statement in the user's CP directory entry. The user automatically becomes a member of the specified group. A user can be assigned membership in multiple groups by specifying multiple group names or GIDs on the POSIXGLIST statement. To avoid ambiguity in the event that multiple groups are defined with the same GID, it is recommended that groups be specified by group name rather than by GID.

Some commands return the name of the user group. These commands include the OPENVM LISTFILE command provided by CMS and the **ls** command provided by the OpenExtensions Shell and Utilities. The group names are defined in the global definitions section of the CP directory using the POSIXGROUP statement.

Selecting Additional Security Features

Additional controls are provided that limit the use of some POSIX features on a system-wide or per-user basis. There is a system configuration file statement that defines attributes and permissions for all users on the system. The QUERYDB specification defines whether the users are ALLOWed or DISALLOWed to query other users' POSIX database information. The EXEC_SETIDS specification defines whether users are ALLOWed or DISALLOWed to have their POSIX IDs changed on behalf of a POSIX exec() function call.

These values can be overridden for a user by the POSIXOPT directory statement. The EXEC_SETIDS option allows the user to execute set-ID programs. This is necessary to run certain programs. The QUERYDB option allows the user to obtain information about groups as well as other users' database information. This is useful in some POSIX functions. The SETIDS option specifies whether the user is authorized to set other users' POSIX IDs. SETIDS should be used only for the BFS servers.

QUERYDB and EXEC_SETIDS DISALLOWed values are ignored for the users with an effective UID of 0. A UID of 0 indicates a user has appropriate privileges. This user is known as a superuser.

The security concern with allowing a user to execute set-ID files is that the user acquires the authority associated with the file. A malicious user could interrupt a program and access data that would normally not be available to this virtual machine.

There are several implications that should be noted if you choose to make use of these additional controls:

- Any user who is not allowed to execute set-ID programs will not be able to use the **mailx** utility provided with the OpenExtensions Shell and Utilities. This utility allows users to exchange notes in a UNIX®-like fashion.
- Any user who is not allowed to query the user database information for other users will not be able to use all of the options of the **ls** utility, which is also provided with the OpenExtensions Shell and Utilities, to find user and group name information related to files.

Creating BFS File Spaces

At the center of OpenExtensions is BFS. It provides the file system interface and semantics required by POSIX. BFS data is managed by a CMS file pool server. A file pool server can manage either SFS or BFS data, or both. For a detailed description of how to set up BFS and how to organize the file system view provided by one or more file pool servers, see *z/VM: CMS File Pool Planning, Administration, and Operation*.

In general, these steps involve allocating storage space to one or more file spaces that are to contain BFS data. These steps define a standard topmost file tree in the system-provided file pool, VMSYS. Once the file system has been set up, you can add the FSROOT value to the POSIXINFO statement of individual users' CP directory entries. The FSROOT value specifies the file system (BFS file space) that should be mounted (made available as the root of the user's directory tree) by CMS when the user starts using OpenExtensions services.

Additional Considerations

The following are additional consideration for working with BFS:

- **Directory MAXCONN value**

  A user virtual machine communicates with the BFS server virtual machine over Advanced Program to Program Communication (APPC) connections. Two APPC paths are used by each POSIX process during its execution and each mounted file system also requires two paths. This means that the user should have a MAXCONN value in the user's CP directory entry of at least 64, and perhaps more if the user's directory tree is composed of elements managed by many filepools.
- **File access in a distributed environment**

  If the BFS server and the user who wishes to access BFS files reside on different VM images, any of the three VM communications servers can be used to provide the connectivity between the systems. However, the POSIX security data provided by CP and sent to the server may be different depending on the communications server used:

  – If the systems are in either a Communication Services (CS) or Transparent Services Access Facility (TSAF) collection (that is, the systems are connected by and running either the Inter-System Facility for Communication (ISFC) or TSAF), the POSIX security values provided are those of the POSIX process that initiated the connection.

  - If the connection is with APPC/VM VTAM® Support (AVS), the values provided are those of the security user at the target system.

- **File processing in a distributed environment**

  If the BFS server and the user wishing to execute a set-ID file reside on two different VM images, those images must be in the same CS collection (that is, the images must be running and connected by ISFC), and the system administrator must ensure that:

  - There is a flat name space for the UIDs and GIDs across the collection. That is, every UID value in the collection must represent a distinct user or set of users, as must every GID.
  - The server where BFS resides is defined as a global resource.
  - All systems in the path between the two systems, as well as the server and requestor systems, must be running a version of CP that supports POSIX.

Part 2. The OpenExtensions Shell

Chapter 2. An Introduction to the OpenExtensions Shell

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

The shell is a command processor that you use to:

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

The Shell Session

A shell user is a CMS user who has logged onto z/VM and has a CMS session. From the CMS command line, the user enters the OPENVM SHELL command. This starts the OpenExtensions Shell, making all the shell commands and utilities available. The shell user can still invoke VM commands with the **cms** command.

There are two categories of shell user: *superuser* and *user*. The superuser has a UID of `0`, can do anything a user can, has special authority to perform certain additional tasks (such as mounting and unmounting a file system), and can access all OpenExtensions services and the files in the byte file system.

The shell user can use the **su** command to switch to superuser authority. To run **su**, your effective UID must be 0, or your effective GID must be 0, or one of your supplementary GIDs must be 0.

The Shell Commands

The OpenExtensions shell provides commands (and utilities) that give the user an efficient way to request a range of services.

POSIX 1003.2 distinguishes between 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). In this document, the term **command** includes both kinds of request.

Shell commands often have *options* (also known as *flags*) that you can specify, and they usually take an *operand*—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 operand, if any. For example:

```
ls -a myfiles
```

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

This book describes various commands you can use to perform certain tasks. Typically, this discussion highlights only certain functions of the command. For complete information about each command and all its options, always refer to *z/VM: OpenExtensions Commands Reference*.

Appendix B lists OpenExtensions 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 CMS OpenExtensions system environment for an application program. Locale affects collation, date and time conventions, numeric and monetary formats, program messages, and yes and no prompts.

The OpenExtensions shell and utilities support any locale generated with code pages IBM-1047, IBM-1027, or IBM-939.

The shell always starts in the POSIX locale, but you can change the locale. See "Changing the Locale: The LC_ Variables" on page 29 for information on changing the locale for the shell and utilities.

Porting Yourself 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 OpenExtensions shell. In particular, the 3270-type terminal interface may surprise you. For example:

OpenExtensions Shell Behavior	For More Information
The 3270 interface operates in line mode. You type data on a command line and no data is transmitted until you press the <Enter> key.	"Understanding the 3270 Screen" on page 17
Instead of using a <Ctrl> key to type control sequences (for example, <Ctrl-D>), you use an escape-key sequence.	Appendix F, "Escape Sequences," on page 203
Certain escape characters are not recognized nor acted upon when they appear within data displayed on the terminal.	"Limitations For Display of Data On the Terminal" on page 18
The OpenExtensions shell uses the EBCDIC code pages IBM-1047 and IBM-1027 for singlebyte data. Any singlebyte data moved into the byte file system (BFS) from an ASCII workstation and some data from an CMS country-extended code page will need to be converted to the code page IBM-1047.	"Understanding Code Page Conversion" on page 20
The left and right square brackets [ ] have different hexadecimal encodings on the shell-supported code pages from what they have on most workstation keyboards (unless you are using an APL character set). To work around this, you have several choices.	"Understanding Code Page Conversion" on page 20
To view help for a CMS or shell command, use the z/VM HELP facility. This will allow you to view individual CMS help files or the task panels where you can pick which help to view.	"Online Help" on page 50
In AIX, entering the exclamation point (!) is the equivalent of typing the **history** command. In the OpenExtensions shell, you use the **history** command or the Retrieve function key.	"Retrieving Previously Entered Commands" on page 47
c89 is available, but not **cc**. Also, **c++** is available as **cxx**.	The **c89/cxx** description in the *z/VM: OpenExtensions Commands Reference*
You can use the **vi** editor at the workstation, but not in the OpenExtensions shell. Working at the host, you can edit BFS files using XEDIT or the **ed** editor.	Chapter 14, "Editing Files," on page 133

OpenExtensions Shell Behavior	For More Information
Many UNIX systems support an executable text file that contains a “magic” number. This is a text file beginning with `#!`*pathname*. For example, you could write a shell script and indicate what shell to use with the first line: `#!/bin/ksh`	The OpenExtensions shell does not support this.
The **mount** and **unmount** commands are available only in CMS, not in the OpenExtensions shell.	“Using Commands to Work with Directories and Files” on page 92

Interoperability

A shell user has access to and can take advantage of the underlying VM system. The shell itself is a CMS application that provides its own command environment. It is started by the user issuing the **OPENVM SHELL** command. From the shell environment, the user can execute any VM command or application residing in the CMS file system by using the **cms** command. The other major areas of interaction between the OpenExtensions facilities and the base CMS environment are:

- From the CMS environment, OpenExtensions applications that reside in the CMS file system can be invoked directly by name. To use an OpenExtensions application that resides in BFS, the user invokes it indirectly by issuing the **OPENVM RUN** command.
- Data can be copied between the CMS file system and byte file system (BFS) by using the **OPENVM GETBFS** and **OPENVM PUTBFS** commands. These commands allow for record-to-byte-stream and code page conversion.
- REXX execs can call shell commands through the **OPENVM RUN** command.
- REXX execs can call OpenExtensions applications through the **OPENVM RUN** command, or if they reside in the CMS file system, directly by name.

Parallels Between the CMS and Shell Environments

An interactive user can make use of both shell and CMS facilities. Programmers whose primary interactive environment is a UNIX or AIX workstation find the OpenExtensions shell programming environment familiar. Those whose primary computing environment is CMS can do much of their work in that environment.

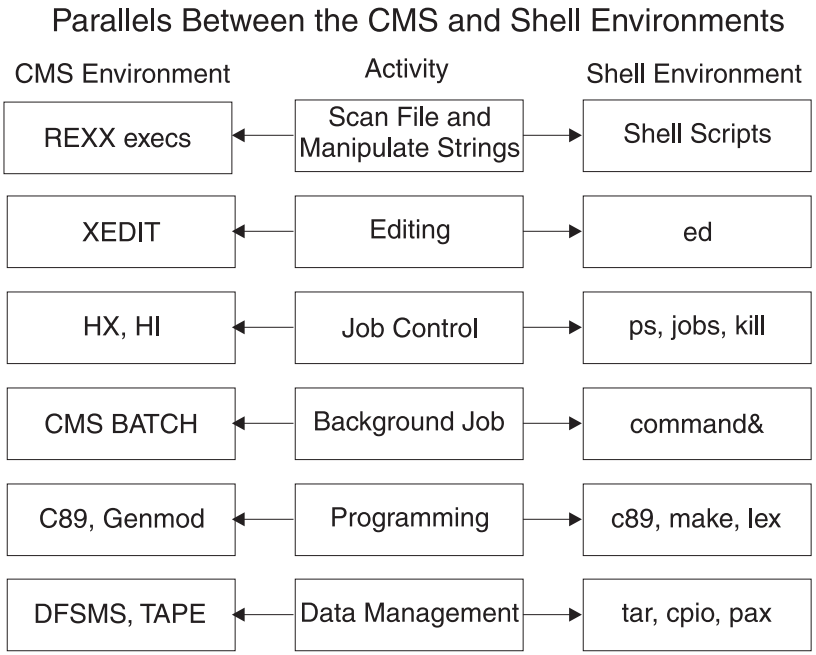


Figure 1. Parallels Between the CMS and Shell Environments

Scanning Files and Manipulating Strings

The shell programming environment provides function similar to the CMS environment with its REXX execs.

REXX is a high-level interpreted language that enables you to write programs in a clear and structured way. You can use REXX to write programs called *REXX programs,* or *REXX execs,* that perform given tasks or groups of tasks. You can run REXX programs that call OpenExtensions services in CMS, in the shell environment, or from a C/C++ program. For more information about writing REXX programs, see *z/VM: REXX/VM User's Guide* and *z/VM: REXX/VM Reference*.

In the OpenExtensions shell, command processing is similar to command processing for execs. You can write executable *shell scripts* (a sequence of shell commands stored in a text file) to perform many programming tasks. With its commands and utilities, the shell provides a rich programming environment.

Editing

In CMS, you edit OpenExtensions byte file system (BFS) files using XEDIT.

Job Control

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

Additionally, in the shell you can use the <EscChar-Z> sequence to stop, or suspend, a foreground job, and you can 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 CMS, you can write a background job and submit it to the CMS batch facility.

Programming

In CMS or in the shell, the **c89** or **cxx** command compiles and builds an OpenExtensions C/C++ program, creating an executable file. In the shell, the **make** command is available for maintaining source and object files, and **lex** and **yacc** are available for developing applications.

Data Management

In CMS, the storage administrator uses the Shared File System (SFS) to automatically back up and archive BFS. There is no facility for backing up individual files.

In the shell, you can use **tar**, **cpio**, and **pax** to read or write an archive file in the file system. There is no automatic backup in the shell.

Archive files can be copied from a BFS directory to a tape or from a tape to a BFS directory using the **OPENVM PARCHIVE** command.

Security

The system programmer defines an OpenExtensions shell user by assigning a user a *POSIX user ID (UID)* and *POSIX group ID (GID)*. The UID and GID are numeric values associated with a VM user ID and set in the CP user directory when a user is authorized to use OpenExtensions services. The system uses the UID and GID to identify the files and processes that a user runs. The UID identifies a user of OpenExtensions 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.

Chapter 3. Using the OpenExtensions Shell

Using CMS

CMS is the interactive and program execution environment provided by VM. CMS reads commands from the terminal and executes the appropriate programs. These programs can in turn execute other programs. Users access CMS through 3270 displays or from workstations through a 3270 emulation application. The OpenExtensions shell and its utilities are CMS application programs. The shell itself is a special kind of CMS application that provides an interactive command environment of its own.

The CMS user issues the command **OPENVM SHELL** to invoke the shell. On many systems, the system administrator will have tailored the CMS initialization process so that this command is executed automatically, thus placing the user in the shell environment upon logon.

Understanding the 3270 Screen

The VM terminal interaction is not a keystroke-reactive environment as is usually the case with UNIX systems. Keystrokes are not read until the **ENTER** key or one of the function keys is pressed. Similarly, the **ALT** and **CTRL** keys cannot be used in combination with other keys to generate signals or other input.

This is the typical appearance of the VM screen in CMS line mode:

```
$
ls -l
total 104
-rw-r--r--   1 bin      bin         1071 May  3 01:18 mailx.rc
-rw-r--r--   1 bin      bin         1457 May  3 01:18 profile.sample
drwxr-xr-x   1 hal      system         0 May  3 01:18 samples
-rwxr-xr-x   1 bin      bin         5007 May  3 01:18 startup.mk
-rw-r--r--   1 bin      bin        11681 May  3 01:18 yylex.c
-rw-r--r--   1 bin      bin        20942 May  3 01:18 yyparse.c
$
id
uid=254(farrell) gid=517(DEPTG37)
$

                                                      RUNNING   GDLVM7
```

At the bottom of the screen, where the cursor can be seen, is the command line. This ″line″ actually wraps around to include the bottom two lines on the screen. This is where commands are entered and prompts are answered. This is also referred to as the *input area*.

Commands that are entered on the command line are passed to the shell for processing. However, if one of the CMS immediate commands (for example, **HX** or **HI**) is entered at the beginning of the command line it is interpreted as an immediate command and is not passed to the shell.

Above the command line and continuing to the top of the screen is the *output area*. This is where the shell prompt is written, command input is echoed and responses are displayed. Nothing can be typed in this area.

In the lower right corner is the *status area*. This is at the end of the command line area and is used by the system to tell the user about the status of the screen. It

includes two parts; the first is the *screen state* and the second is the *system ID*. The screen state changes regularly to assist the user in understanding what is happening in the CMS session. There are five screen states that can be displayed.

Running
The Running state means that the shell is executing and commands can be entered.

More...
When the output area is full, this state is displayed. This indicates that the screen should be cleared so more output can be displayed. You can clear it by pressing the **CLEAR** or **PA2** key. The **PA2** key will clear the output area but leave the input area intact. The **CLEAR** key will clear both the input and output areas. If you do nothing the output area will be cleared after a period of time defined by your system administrator. This period is usually about 30 seconds. If you press the **ENTER** key when this status is displayed, the status will change to **HOLDING** and the screen will not be cleared until you explicitly clear it.

Holding
This means you pressed **ENTER** in response to a **More...**. Press **CLEAR** or **PA2** to clear the screen.

Not Accepted
The system is busy (generally with work requested of the z/VM Control Program component) and cannot process your input. You will most likely see this only if you issue Control Program commands from the shell. Wait until the previously-entered command completes and then reenter your command. This **NOT ACCEPTED** status will be displayed for only three seconds.

VM READ
The virtual machine is attempting to perform a line-mode read from the virtual console. The user must supply a line of input and press **ENTER**. When **ENTER** is pressed, the virtual machine returns to **Running** state.

The second part, the *system ID*, does not change. It is generally used to identify the name of the node in the network by which this system is known.

CMS can be run in a different mode in which the output area is actually both an input and output area and in which there are no screen states. This is called *CMS fullscreen mode*. This document assumes the user is running in CMS line mode, as described above. Users interested in running in CMS fullscreen mode should consult *z/VM: CMS User's Guide*. Note, however, that when the shell is running in CMS fullscreen mode, the status window will display *Running a Command* and the message indicator will not be accurate.

Limitations For Display of Data On the Terminal

The following escape characters are not recognized nor acted upon when they appear within data displayed on the terminal:

- \a - sounding of printer bell
- \f - formfeed
- \v - vertical tag
- \b - backspace

Many utilities have options, for example the **-f** option of the **pr** utility, that makes it possible to include these characters in output displays and files.

The lack of proper display of these characters does not indicate a lack of utility support or the omission of the escape characters in the file or display.

The Shell Run-time Requirements

The OpenExtensions shell commands and utilities are all C/C++ applications and as such require the C/C++ run-time library to be available on an accessed minidisk or SFS directory. SCEERUN LOADLIB must also be in the **GLOBAL LOADLIB** list. To facilitate this, the **OPENVM SHELL** command automatically establishes the C/C++ run-time environment based on statements it finds in `/etc/openvmdefaults`.

There are two types of statements in the `/etc/openvmdefaults` file. The **CLINKNAME** statement specifies the minidisk or directory that contains the C/C++ library. This statement contains the keyword **CLINKNAME** followed by the disk name. The disk name can be anything that is supported by the **VMLINK** command. If the disk name is a directory, the directory name must be preceded by the keyword **.DIR**. The other is the **CLIBNAMES** statement which specifies the name of one or more load libraries to be appended, in that specific order, to the **GLOBAL LOADLIB** list. The format of this statement is the keyword **CLIBNAMES** followed by one or more load library names.

For example, if the C/C++ library resides on user MAINT's 1A1 disk and the load library is the standard SCEERUN, the `/etc/openvmdefaults` file would look like this:

```
CLINKNAME MAINT.1A1
CLIBNAMES SCEERUN
```

Statements in `/etc/openvmdefaults` are delimited by the default newline character (X'15'). The keyword must be in upper case and must be the first non-blank word on the line. Multiple statements of this type are allowed.

If `/etc/openvmdefaults` does not exist, **OPENVM SHELL** will append SCEERUN to the **GLOBAL LOADLIB** list, it if is found in the search order. If not, the **OPENVM SHELL** command will be unsuccessful.

Exiting the Shell

When you have finished using the shell you can exit to CMS or log off from the VM system. To exit to CMS, enter the shell **exit** command. However, this exit command will not be processed until all background jobs complete. To log off, enter the **LOGOFF** command with the **cms** command, (that is, enter **cms logoff**). This will end your CMS session regardless of the state of any background job and will stop any background job that is running.

Getting Rid of a Hung Application

If your application hangs, try the following procedure to get rid of it:

1. On the command line, enter <EscChar-V> or <EscChar-C>. When this is successful, the shell prompt is displayed.
2. If the previous step does not work, the next step is to halt the shell by issuing the **HX** command. If this works, CMS abnormal termination messages will appear and the terminal will enter VM READ state. Enter the **BEGIN** command to continue.

3. The last choice is to restart the CMS session. Press the **PA1** key and then enter **IPL**. If this short form of the **IPL** command is not accepted (a consequence of how CMS was set up on your system) issue **IPL CMS** to restart CMS.

Understanding Code Page Conversion

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.

For internal processing, the OpenExtensions shell and utilities and many other POSIX programs can operate only on data in the byte file system (BFS) that is encoded in one of three supported code pages:

- IBM-1047
- IBM-1027
- IBM-939

(The charmaps for these code pages are in `/usr/lib/nls/charmap`.)

Data in any other code page (for example, coming from a CMS record file or a workstation) must be converted to one of the supported code pages before it can be processed. The three supported code pages are compatible with the C/C++ compiler on the encodings for characters in the POSIX portable character set.

The following 13 characters in the POSIX portable character set may have hexadecimal encodings that differ between the shell-supported code pages and the country-extended code page used by VM in your system:

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

To see the POSIX portable character set and code page 00293 used by the C/C++ compiler, turn to Appendix E.

Customizing the Square Brackets on Your Keyboard

Between the shell-supported code pages and any other code page, an application programmer needs to be concerned about two characters in the POSIX portable character set that will have different encodings:

Left square bracket: [
Right square bracket:]

If you do not use an APL character set, on many programmable workstations you can customize your keys so that you have hexadecimal encodings for the left and right square brackets that match the shell-supported code pages:

X'AD' for a left square bracket ([)
X'BD' for a right square bracket (])

If you cannot program your workstation terminal emulator to produce these encodings, then you can use CMS's SET INPUT and SET OUTPUT commands to accommodate your emulator. To customize the square brackets, use these commands:

If the square brackets are not being entered correctly, enter:

```
SET INPUT [ AD
SET INPUT ] BD
```

If the square brackets are not being displayed correctly, enter:

```
SET PUTPUT [ AD
SET OUTPUT ] BD
```

When Do You Need to Convert between Code Pages?

You need to convert from one code page to another when:

- Transferring files between a workstation and the file system
- Copying data between CMS record files and BFS files
- Converting between ASCII and EBCDIC when using the **pax** utility

There are several options for converting data to or from a shell-supported code page:

- To convert singlebyte data that contains the square brackets when you are moving the data between VM and the byte file system, you can use the **(TRANSLATE** option on the **OPENVM PUTBFS** and **OPENVM GETBFS** commands. The **(TRANSLATE** option does not convert doublebyte data.
- To convert doublebyte or singlebyte data to a selected code page while you are working in VM, use the C/C++ **iconv** utility. For information on how to use this utility, see the *XL C/C++ for z/VM: User's Guide*. Note that a code page is also known as a code set.
- To convert doublebyte or singlebyte data to a selected code page while you are working in the shell, use the **iconv** shell command.

Naming Files Using the POSIX Portable File Name Character Set

To simplify conversion requirements, you should use the POSIX portable file name character set when naming your files:

Uppercase A to Z
Lowercase a to z
Numbers `0` to 9
Period (`.`)
Underscore (_)
Hyphen (-)

Default Escape character and LINEDEL

This default escape character may conflict with the LINEDEL character in effect for the user's terminal. A shell user should enter `cms terminal linedel off` at the shell prompt or should enter `TERMINAL LINEDEL OFF` before invoking the shell. This command is a good candidate for inclusion in the `.profile` file.

In this document, references are made to the escape character, which by default is the cent sign (¢). For example, on a UNIX system you would provide an end-of-file to a program by typing the **ALT** key combined with the **D** key. As mentioned above,

such key sequences are performed in the shell by typing the escape character followed by a key. An end-of-file would be entered by typing **¢C** followed by pressing the **ENTER** key.

Default CP Terminal Escape Character and the Shell

The default CP terminal escape character is the " (double quotation mark). This may interfere with some shell commands, such as **awk**.

To avoid the interference, change the escape character to some other character by using the **CP TERMINAL ESCAPE** command. The character you select for your CP escape character should not appear in the data you are entering.

Another way to avoid interference is to disable CP's checking for terminal escape characters. To do that, use the command **CP TERMINAL ESCAPE OFF**. When you are ready to enable checking again, use the command **CP TERMINAL ESCAPE ON**.

Default CP Terminal Line End Character and the Shell

The default CP line end character is the # (number sign). This may interfere with some shell commands, such as **awk**.

To avoid the interference, change the line end character to some other character by using the **CP TERMINAL LINEND** command. The character you select for your CP line end character should not appear in the data you are entering.

Another way to avoid interference is to disable CP's checking for line end characters. To do that, use the command **CP TERMINAL LINEND OFF**. When you are ready to enable checking again, use the command **CP TERMINAL LINEND ON**.

Chapter 4. Customizing the Shell

You can personalize your use of the OpenExtensions shell. This chapter discusses:
- Creating or modifying your `.profile` file
- Understanding environment variables
- Customizing your shell environment with the *ENV* variable
- Customizing the search path for commands with the *PATH* variable
- Setting options for a shell session
- Customizing your shell interface

Customizing Your .profile

When you start the OpenExtensions shell, it uses two levels of environment variables to meet your particular needs or preferences as a user. The first level is a default systemwide user environment that is established when the shell executes `/etc/profile`, the system-wide login script for the shell. The system programmer may modify the variables in this file to reflect local needs (for example, the time zone). If you do not have an individual user profile, the values in `/etc/profile` are used during your shell session.

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

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

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

export PATH EDITOR PS1      #export global variables
```

Figure 2. 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 quotation marks.

ENV=$HOME/.setup
: Identifies `.setup` in your home directory as your login script. See “Customizing Your Shell Environment: The ENV Variable” on page 27 for more information about a login script.

export ENV
: Specifies whenever a subshell is created, the *ENV* variable should be exported to it. See “Exporting Variables” on page 58 for more information about exporting variables.

PATH=$PATH:$HOME:
: Identifies the search path to be used when locating a file or directory. Here, the system first searches the path identified in the *PATH* variable in

`/etc/profile`, the system profile; it then searches your home directory. See "Customizing the Search Path for Commands: The PATH Variable" on page 28 for more information.

PS1='$LOGNAME':'$PWD':' >'
Identifies the shell prompt that indicates when the shell is ready for input. 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.

EDITOR=ed
Identifies **ed** as the default editor used by some of the utilities, such as **mailx**.

export PATH EDITOR PS1
Specifies whenever a subshell is created, these variables should be exported to it. See "Exporting Variables" on page 58 for more information about exporting variables.

If you create a subshell with the command **sh –L**, the shell starts and reads and processes your profile file. The shell looks for `.profile` 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 the value in quotation marks. The quotation marks tell the shell to treat blanks as literals and not delimiters. Single quotation marks are more "serious" about this than are double quotation marks:

- Single quotation marks preserve the meaning of (that is, treat literally) all characters.
- Double quotation marks 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 quotation marks, 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 Environment Variables

You can display the shell's environment variables and their values by using the **set** command. You may see many variables that you do not recognize. These are *built-in*, or *predefined*, variables that are set up with default values when you start the shell. In other parts of this book, some of these predefined variables are discussed; for complete information, see the **sh** command description in *z/VM: OpenExtensions Commands Reference*.

You can display the value of a single variable with the **echo** command. For example:

```
echo $HOME
```

displays the current value of the *HOME* variable.

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

```
echo *.doc
```

The shell first expands the wildcard character *****. This construct `*.doc` 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. If there are no filenames ending in `.doc`, the command output is just `*.doc`.

Table 1 lists some of the more frequently used built-in variables. For more information on all the built-in variables, see the **sh** command description in *z/VM: OpenExtensions Commands Reference*. You can customize the values of many of these variables by using your `.profile`; be aware, though, that only `.profile`; only *IFS*, *PS1*, and *PS2* support doublebyte characters for the values.

Table 1. Built-in Variables

Variable	Purpose
_	(Underscore) expands to the last operand from the previously processed command. For every command that is processed as a child of the shell, **sh** sets this variable to the full path name of the executable file and passes this value through the environment to that child process. When processing the *MAILPATH* variable, this variable holds the value of the corresponding mail file.
CDPATH	Contains a list of directories for the **cd** command to search. Directory names are separated with colons. *CDPATH* works in a similar way to the *PATH* variable.
COLUMNS	Used by several commands to define the width of the terminal output device.

Table 1. Built-in Variables (continued)

Variable	Purpose
EDITOR	Specifies the default editor. This variable is usually set in your `.profile`.
ENV	**sh** performs parameter substitution on this value and uses the result as the name of an initialization file, or login script. This file is processed with the **.** (**dot**) command; see the **dot** command in *z/VM: OpenExtensions Commands Reference*. See "Customizing Your Shell Environment: The ENV Variable" on page 27 for more information about the *ENV* variable. This variable is usually set in your `.profile`.
FCEDIT	Contains the name of the default editor for the **fc** command. If this variable is not set, the default is the **ed** command.
HISTFILE	Contains the path name of a file to be used as the history file. When the shell starts, the value of this variable overrides the default history file. See "Retrieving Commands from the History File" on page 47.
HISTSIZE	Contains the maximum number of commands that the shell keeps in the history file. If this variable contains a valid positive integer when the shell starts, it overrides the default of 127.
HOME	Contains the path name of your home directory. This is also the default directory for the **cd** command.
IFS	Contains a series of characters to be used as *internal field separator* characters. Any of these characters can separate operands in unquoted command substitutions such as `` `command` `` or `$(command)`, or in parameter substitutions. In addition, the shell uses these characters to separate values put into variables with the **read** command. Finally, the first character in the value of *IFS* separates the positional parameters in `$*` expansion. See "Using Parameter and Variable Expansion" on page 62 for more information on positional parameters.
LANG	Contains the default locale value.
LC_ALL	Indicates the locale to be used to override any values for locale categories specified by *LANG* or any of the *LC_* variables, such as *LC_COLLATE*, *LC_CTYPE,* and *LC_MESSAGES*, which a user can set and interrogate.
LINENO	Contains the number of the line currently being processed by a shell script.
MAIL	Contains the path name of your system mailbox. If the *MAILPATH* variable is not set, the shell tells you when new mail arrives in this file. The shell assumes that new mail has arrived if the file modification time changes.
MAILCHECK	Contains the number of seconds of elapsed time that must pass before the system checks for mail; the default value is 600 seconds. When using the *MAIL* or *MAILPATH* variables, the shell checks for mail before issuing a prompt.
MAILPATH	Contains a list of mailbox files. This overrides the *MAIL* variable. The mailbox list is separated by colons. If any name is followed by **?***message* or **%***message,* **sh** displays the message if the corresponding file has changed. **sh** performs parameter and command substitution on *message,* and the variable _ (temporarily) expands to the name of the mailbox file.
MBOX	Contains the path name of your personal mailbox, usually `$HOME/mbox`, used to store messages that have been read from your system mailbox. This variable is usually set in your `.profile`.

Table 1. Built-in Variables (continued)

Variable	Purpose
OLDPWD	Contains the name of the directory you were previously working in. The **cd** command sets this variable.
PATH	Contains a list of directories that the system searches to find executable commands. Directories in this list are separated with colons. **sh** searches each directory in the order specified in the list until it finds a matching executable command. If you want the shell to search the working directory, put a null string in the list of directories (for example, to tell the shell to search the working directory first, start the list with a colon or semicolon).
PS1	Contains the primary prompt string used when the shell is interactive. The default value is `$ `. The shell expands parameters before the prompt is printed. A single exclamation mark (!) in the prompt string is replaced by the command number from the history list; see **fc** in *z/VM: OpenExtensions Commands Reference*. For a real exclamation mark in the prompt, use !!. This variable is usually set in your `.profile`.
PS2	Contains the secondary prompt, or continuation prompt, used when completing the input of such things as reserved word-commands, and quoted strings. The default value of this variable is `> `.
PWD	Contains the name of the working directory. When the shell starts, the working directory name is assigned to *PWD* unless the variable already has a value.
SECONDS	Contains elapsed time. The value of this variable increase by 1 for each elapsed second of real time. Any value assigned to this variable sets the *SECONDS* counter to that value; initially the shell sets the value to `0`.
SHELL	Contains the full path name of the current shell.

Customizing Your Shell Environment: The ENV Variable

The customization discussed so far is set up inside your `.profile` file. However, the shell reads your `.profile` only when you start the shell or when you enter the **sh** command with the **–L** option.

To 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*. 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 enter the **OPENVM SHELL** command and whenever you explicitly create the shell during a session (for example, as a subshell to run a shell script). To make sure *ENV* is set up after you enter the OPENVM SHELL command, put *ENV* into your `.profile` file. For example:

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

where `.setup` is the name of your login script.

To make sure *ENV* is set up when you execute a subshell, put this into your `.profile` file after the *ENV* statement:

```
export ENV
```

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 one time.

Customizing the Search Path for Commands: The PATH Variable

Command interpreters usually have to *search* for a file that implements 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 OpenExtensions shell commands are stored) before the `/usr/bin` directory.

If you set up your *PATH* incorrectly, you could get the wrong command. You should probably always search the shell commands directory first: `/bin`. Some OpenExtensions shell commands run other shell commands and utilities by name; they expect to get the OpenExtensions 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
```

You should be careful when including the working directory early in your search path. If the working directory includes a copy of an executable file that is also found in /bin, then the working directory's copy will be executed instead of the copy from /bin. If some other user knows you have included the working directory in your path, then he could exploit this fact to trick you into running a Trojan horse. Be careful to watch for such situations if you choose to include the working directory early in your search path.

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 processed when you enter a command. The **type** command can tell you which file is processed 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
```

Using **type** you can figure out how the search path works and what effect aliases have.

Changing the Locale: The LC_ Variables

To set the locale you want, you set the value for the LC_ALL variable and export it. This variable overrides any values for locale specified with LANG or any of the LC_ variables (such as LC_COLLATE and LC_MESSAGES) *except* LC_CTYPE.

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** 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, follow it with **exec sh**. For example, to work in the French Canadian locale, add this to your `.profile` file:

```
LC_ALL=Fr_CA
export LC_ALL
exec sh
```

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 *z/VM: OpenExtensions Commands 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 Chapter 5 for more information.

Preventing Wildcard Character Expansion

The command:

```
set -o noglob
```

tells the shell not to expand wildcard characters in file names. 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 File Names” on page 45 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.

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. Working with Shell Commands

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

Specifying Shell Command Options and Operands

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 100 for an example.)

```
ls –1
```

lists directory contents 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 operands for the command (that is, operands 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 operands must be typed as singlebyte characters. Additionally, delimiters such as slashes, braces, and parentheses must be typed as singlebyte characters.

The order of options and operands 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 OpenExtensions shell commands let you specify a double minus sign (– –) to separate the options from the nonoption operands; – – 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 OpenExtensions 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 *z/VM: OpenExtensions Commands Reference*.) For example,

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 **cat –tv file**.

Specifying Options with Accompanying Operands

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

For example, if you read the description of **ps** in *z/VM: OpenExtensions Commands Reference*, you will see that **ps** accepts an operand of the form:

```
–u userlist
```

When *z/VM: OpenExtensions Commands 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, *userlist* should be a string of one or more UID numbers or login names separated by commas and enclosed in single quotation marks. In the command:

```
ps –u 'macneil,wellie1'
```

userlist is `macneil,wellie1`. (If the string does not contain spaces, tabs, or other special characters, you can actually omit the enclosing single quotation marks, but the command is often easier to read if you use quotation marks anyway.) When it runs, **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, use the z/VM HELP facility (see “Online Help” on page 50).

Interrupting a Shell Command

If you want to interrupt a command and stop it from completing, type <EscChar-C>.

Understanding Standard Input, Standard Output, and Standard Error

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

It reads from its *standard input* file. By default, standard input is the keyboard.

It writes to its *standard output* file. By default, standard output is the screen.

It writes error messages to its *standard error* file. By default, standard error is the screen.

In the OpenExtensions 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 *z/VM: OpenExtensions Commands Reference*.

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 OpenExtensions 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`.

Redirecting Input from a File

You can redirect input in much the same way that you redirect output. A command that usually 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`, rather than your input from the keyboard, becomes input to **mailx**.

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 shell 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 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 OpenExtensions 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 usual 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/.profile`). When you issue the **OPENVM SHELL** command or start a shell with **sh –L**, the shell reads the aliases from the file and sets them up immediately. See "Customizing Your .profile" on page 23 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 a never-ending 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"
```

and the command that ends up being called is:

```
typeset -f abc
```

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.

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`. Usually, **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 file name 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 OpenExtensions 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 path name for a command.

To use alias tracking, enter the command:

```
set -o trackall
```

The first time you enter a command, the shell creates an alias that is the full path name of the command. For example, if you entered the **ps** command, the shell would create the alias:

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

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

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 (this does not apply to tracked aliases). For example:

  `\rm` *`file`*
- Specify the full path name. 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 path name.

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 **&&**
  Using **||**
- 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 OpenExtensions 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
```

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

Using && or ||

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 OpenExtensions 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 *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 `` ` ` `` (accent grave) 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 file names: `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* 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 file names 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 OpenExtensions shell version of **find** 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 *z/VM: OpenExtensions Commands 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.

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; lets you enter more than one command on the same line.
( )	In a sequence of commands, groups those commands that are to run as a separate process in a subshell. This means the current shell invokes a second shell and the second shell actually runs the command. Thus, the command runs in a separate execution environment: It can change working directories, change variables, open files, and so on, without affecting the first shell. **( )** is also used to group mathematical operations.
{ }	In a sequence of commands, groups commands and runs the command in braces *without* creating a subshell. 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.

Used in File Names

Character	Usage
/	Separates the parts of a file's path name.

~	(Tilde) symbolizes the home directory when used at the beginning of a file name. For example,
	`~/.profile`
	
	refers to the user'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 in which were previously working.

.	By convention, indicates that what follows the **.** is an extension, or suffix, to the file name.
?	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 35.
>	Redirects output to a specified file.	"Redirecting Command Output to a File" on page 35.

Character	Usage	Example
>>	Redirects output to be appended to the end of the specified file.	“Redirecting Command Output to a File” on page 35.
2>	Redirects error output to a specified file.	“Redirecting Error Output to a File” on page 36.

<<EOF	Redirects input until EOF.	This is used in a “here-document.” For example, say you had a shell script named *here*:
		`ls <<EOF`
		`/bin`
		`EOF`
		
		When you run the shell script, it runs the **ls** command using the parameters between the two EOFs.

Using a Special Character Without Its Special Meaning

If you do not want to use the special sense of a metacharacter you can turn off the special meaning with any of these constructs:

```
\
' '
" "
```

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 Quotation Marks (' ')

A pair of single quotation marks (' ') turns off the special meaning of *all characters* within the quotation marks. (An apostrophe ’ character is treated the same as a single quotation mark.)

A Pair of Double Quotation Marks (" ")

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

Using a Wildcard Character to Specify File Names

If you have used other operating systems, you are probably familiar with the concept of *wildcard characters*. (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 file names in shell commands.

The shell replaces the pattern with the list of matching files before the command is invoked. The OpenExtensions shell recognizes several different wildcard characters:

*****
?
[ ]

The * Character

The asterisk (*****) stands for any sequence of zero or more characters. You can use the asterisk in file names. For example:

```
ls aa*
```

lists all files in the working directory with names that begin with `aa`. The shell translates the command into `aa.c aa.o aa.output .`

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 file names. 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 path name, the question mark **?** can stand for any single character. 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.

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

```
ls *.?
```

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

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

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

shows all files in every directory under your working directory that has 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]
```

Matches any character from `a` through `m`. This does not include hexadecimal ranges.

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 use a 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 OpenExtensions shell, there are two ways to retrieve previously called commands:

- the VM retrieve function key
- the **history** command, combined with the **r** command

Using the Retrieve Keys

Retrieve keys provide a way to recall previously issued commands to the command line, where they can be edited and reissued. This key is usually defined in the user's PROFILE EXEC, but can be assigned at anytime using the **SET RETRIEVE** command. To set the **F12** key to be the **RETRIEVE** key, you would would enter:

```
set pf12 retrieve
```

This is a VM command (specifically a control program command) so to enter it from the shell, prefix it with **cms**.

Press the **RETRIEVE** key repeatedly until the command you want to use is displayed on the command line. After the command is displayed, you can modify the command or use it as it is displayed. Press **ENTER** to run the command.

Retrieving Commands from the History File

The shell uses a file in your home directory to record each command you enter. 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 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 one time.

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 **c89** command to compile your program. Then:

```
r c89
```

looks back through the history and runs the most recent **c89** 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 usual 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 each one 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 a new file name each time.

You can still do this using the history file. The command:

```
r old=new command
```

runs a previous command but replaces the first occurrence of *old* with *new*. 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 did not set up; for example:

```
history="fc -l"
```

The command **history** is actually a *predefined* alias for the **fc** command with the **–l** option. The **fc** command displays and edits 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 is easier to use and read than the straight **fc** command. For full details about **fc**, see *z/VM: OpenExtensions Commands 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 103.)

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 *name*. Each line is labeled with the name of the file that contains the line. Using this technique 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. 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 command **cksum** 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. Be careful, however, because some changes are not detectable with certain checksum schemes employed by **cksum**.

The command **find** 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 regular expression *regexp*. (For further information, see the appendix on regular expressions in *z/VM: OpenExtensions Commands Reference*.) For example:

```
awk '/abc/ {print}' file
```

displays every record in `file` that contains the string `abc`.

For more discussion on using **awk**, see Appendix C.

Timing Programs

The command **time** 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 OpenExtensions system services for the user, labeled `sys`

Online Help

The CMS **HELP** command provides help for reference information about commands. (If you are entering the command from the shell, remember to preface the commands with **cms**.)

If you are unfamiliar with the HELP facility, you can enter:

```
help
```

to display the main HELP menu. The menu displayed will look similar to:

```
HELP TASKS                Task Help Information          line 1 of 24
(c) Copyright IBM Corporation 1990, 1995

Move the cursor to the task that you want, then press the ENTER key
or the PF1 key.


TASKS     - Help if you don't know z/VM commands.
            Good choice for beginners
MENUS     - List the HELP component MENUs
HELP      - Explain some ways for using HELP
COMMANDS  - List z/VM commands that you can use
CMS       - Show only CMS commands
CP        - Show only CP commands
OPTIONS   - Show options for the QUERY and SET
            commands of both CMS and CP
SUBCMDS   - List z/VM subcommands that you can
            use, such as XEDIT
STATEMTS  - Show statements for REXX, EXEC 2, and EXEC
PF1= Help     2= Top      3= Quit     4= Return     5= Clocate   6= ?
PF7= Backward 8= Forward  9= PFkeys  10=           11=          12= Cu
```

Or, for more information on the HELP command, enter the following command:

```
help cms help
```

to see the following panel:

```
CMS HELP                 Brief Help Information              line 1 of 12

Brief Information

The HELP command tells you how to use z/VM HELP.  For introductory
information on the z/VM HELP Facility itself, enter HELP SELF.

FORMAT:  Help component-name command-name (options

EXAMPLE: You forgot how to use the ACCESS command.  If you want to
         get help for that command, then enter:
                        help cms access







PF1= All      2= Top      3= Quit     4= Return     5= Clocate   6= ?
PF7= Backward 8= Forward  9= PFkeys  10=           11=          12= Cursor
```

Then press the **PF1** key to display the syntax for the **HELP** command. Then you will see:

```
 CMS HELP                All Help Information             line 1 of 720
(c) Copyright IBM Corporation 1995

 HELP

           .-TASKs--------.
 >>--Help--+--------------+-------------------------------------------
           |-| Choice A |-|
           |-| Choice B |-|
           '-| Choice C |-'

 Choice A:
    .-TASKs-------------------------.
 |--+-Help-------------------------+---------------------------------
    |-taskname--TASKs--------------|
    |-menuname--MENU---------------|
    '-.----------------.--cmd_name-'
      '-component_name-'

 PF1= Brief    2= Top      3= Quit     4= Return     5= Clocate   6= ?
 PF7= Backward 8= Forward  9= PFkeys  10=           11=          12= Cursor
```

For more information about the HELP facility, see *z/VM: CMS User's Guide*; for more about the **HELP** command, see *z/VM: CMS Commands and Utilities Reference*.

Example: Getting Help for OPENVM Commands

For example, if you want information on all the CMS OPENVM commands, you would enter:

```
cms help openvm
```

The output displayed looks similar to:

```
 OPENVM MENU               Menu Help Information                 line 1 of 16
 (c) Copyright IBM Corporation 1990, 1995

 The file names listed below represent CMS OPENVM commands.

 A file may be selected for viewing by placing the cursor under any
 character of the file wanted and pressing the ENTER key or the PF1 key.
 A menu file is indicated when a name is preceded by an asterisk (*).
 A task file is indicated when a name is preceded by a colon (:).
 For a description of the HELP subcommands and options, type HELP HELP.


 *CMS      CREDIREC  ERASE     PARchive  PERMit    QMASK     SETDIREC
 *OMACRO   CREEXTLI  GETbfs    PATHCREA  PUTbfs    QMOUNT    SETMASK
 *OROUTINE CRELINK   LIStfile  PATHDELE  QDIRECTO  REName    SHell
 *OSHELL   CRESYMLI  MOUnt     pathname  QLINK     RUN       UNMount
 *XEDIT    DEBUG     OWNer     PATHQUER
* * * End of File * * *

 PF1= Help     2= Top      3= Quit     4= Return     5= Clocate   6= ?
 PF7= Backward 8= Forward  9= PFkeys  10=           11=          12= Cursor

====>
```

Example: Getting Help for OSHELL cp

For example, if you want information on the **cp** shell command from the shell, you would enter:

```
cms help oshell cp
```

The output displayed looks similar to:

```
 OSHELL CP                All Help Information                  line 1 of 109
(c) Copyright IBM Corporation 1993, 1995

 CP -- COPY A FILE

 cp (-fimp) file1 file2
 cp (-fimp) file ... directory
 cp -R (-fimp) source...  directory
 cp -r (-fimp) source...  directory

 Purpose

 cp copies files to a target named by the last argument on its command line.
 If the target is an existing file, cp overwrites it; if it does not exist, cp
 creates it.  If the target file already exists and does not have write
 permission, cp denies access and continues with the next copy.

 If you specify more than two path names, the last path name (that is, the
 target) must be a directory.  If the target is a directory, cp copies the
 PF1=          2= Top      3= Quit     4= Return     5= Clocate   6= ?
 PF7= Backward 8= Forward  9= PFkeys  10=           11= Related  12= Cursor

====>
```

Chapter 6. Writing Shell Scripts

PI

Most people find themselves using some sequences of commands over and over again.

- A programmer might always use the same commands to compile source code and link the resulting object code.
- A bookkeeper might have to go through the same sequence of 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 made "executable" ("processable"), the programmer can run all the commands in the file by entering the file name 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 one time. 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 do the job. 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 commands in it. If you enter:

```
totals.scp
```

the shell runs the three commands.

After you create a shell script, you need to be sure that the file containing the shell script has read and execute permissions. Use the **chmod** and **umask** commands to set the permissions. See the discussion of permissions in Chapter 12 and the command descriptions in *z/VM: OpenExtensions Commands Reference*.

Note: By default, a shell script does not inherit any variables from your current shell session. To pass on a variable, you must export it.

For another example, suppose you want to compile and link a program composed of a collection of files written in the C programming language. 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**.

Additionally, the command:

```
sh file
```

runs a shell script stored in the specified file. The **sh** command starts a new shell to run a command.

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 usual 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 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* variables.

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 quotation marks. 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 usual 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`, integer part thereof
`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 quotation marks around the expression. Thus:

```
let "i=5+2*3"
```

assigns `11` to *i*, because 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, the examples have been 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 variable; enter:

```
export VAR
```

The **export** command says that you want the variable *VAR* passed on to all the commands and shell scripts that you process 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. You could also combine these two commands into a single command:

```
export MYNAME="Robin Hood"
```

Note: You could use single or double quotation marks to enclose the variable value. See “Quoting Variable Values” on page 24 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.

By default, any variables created within a shell script are *local* to that script. When another program is run, those variables do not appear 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; the calling shell does not see them.

Some variables are automatically marked for export by the software that creates them. For example, if you call the shell, the initialization procedure automatically marks the *HOME* variable for export so that other commands and shell scripts can use it. Other variables must be exported 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 operands 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 **$***N* construct (that is, a **$** followed by a number), 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 `echoit` consisting only of the command:

```
echo $1
```

Suppose this command is run:

```
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 operands*.

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 `Hello`.

Positional parameters that include a blank can be enclosed in quotation marks (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`.

If the value of the *$N* parameter includes special characters, the shell ignores those special characters (that is, it treats them literally) when it evaluates the line in which the value was inserted. Returning to the `lookup` example, if you enter:

```
lookup 'abc"def'
```

the parameter value `abc"def` replaces the construct *$1* in the **grep** command, but the ″ is treated literally; **grep** runs as if you had entered

```
grep abc\"def *
```

where **** is the escape character.

Using Quotation Marks to Enclose a $N Construct in a Shell Script

A **$***N* construct in a shell script can be enclosed in double or single quotation marks.

- When *double* quotation marks 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 had not contained the double quotation marks, the parameter replacement would have resulted in:

```
grep two words *
```

which has an entirely different meaning.

- When you use *single* quotation marks 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 quotation marks are “stronger” than double quotation marks. Less is more!

Using Parameter and Variable Expansion

As was shown, 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 ${15}
```

appeared in a shell script, it would echo the fifteenth 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 OpenExtensions 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. As all the following examples suggest, these parameter modifiers are intended to let you break off parts of file names.

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 *value*. 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 *parameter* does not currently have a value, then *value* is assigned to *parameter*, and then the new value of parameter is used. Thus the *:=* form actually assigns a value if *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 *parameter* is null, *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 *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 *replacement* 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 called 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 *pattern*. For example, suppose that the variable *NAME* stands for a file name. You might use:
	`${NAME#*/}`
	
	to remove the highest-level directory from the path name. 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 *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 OpenExtensions shell has a variety of special parameters that may be used in command lines and shell scripts.

Parameter	Meaning

$@	The complete list of positional parameters, each separated by a single space. For example:
	`echo "$@"`
	
	If the positional parameters are all file names:
	`cp "$@" dir`
	
	copies all the files to the given directory `dir`. By using the double quotation marks, the command will be interpreted correctly if the parameter contains blanks.

$*	The complete list of positional parameters, each separated by the first character of the value of the shell variable *IFS*. For example, with:
	`IFS=,$IFS`
	
	then:
	`echo "$*"`
	
	displays the parameters with separating commas.

$#	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 *z/VM: OpenExtensions Commands 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=1* is the equivalent of *[1=1]*.

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 that 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 *pathname* is a directory, this shell script uses **echo** to display an appropriate message and then uses **ls** to display a listing of its contents. Otherwise the shell script assumes *pathname* is a file and echos an appropriate message and then uses **cat** to display the file itself.

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 file name; 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 *commands*. The shell then goes back to check *condition*. If it is still true, the shell runs *commands* again, and so on, until *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 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.)

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 *commands* one time. Then the shell assigns the next string in *list* to *name*, and repeats *commands*. The shell runs *commands* one time 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 file name 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")
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")
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 file name specified on the command line.

Using Functions

A *shell function* is similar to a subroutine in other programming languages: 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 "$1"               # test if first operand is directory
    then
        curdir=$(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 one or two operands. 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 do 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.

Variables set prior to the call are visible to and can be changed by the called function, and variables created by the called function are visible to and can be changed by the caller.

PI end

Chapter 7. Using Job Control in the Shell

When you enter a shell command, you start a *process* in which the command runs. When the process completes, the system displays the shell prompt. When you enter that command, the OpenExtensions 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.

When you enter a shell command, 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 virtual-machine-wide 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 (in the user's virtual machine) 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 signals 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**.

Running Several Jobs at the Same Time (Foreground and Background)

The OpenExtensions shell can run more than one job at a time, and one can be running in the foreground while another is running in the background.

If you type a command and press **ENTER**, you see the output from the command displayed on your screen. You cannot enter any other commands until the shell prompt ($) 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.

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.

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.

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 a foreground process. Therefore, if a program reads from *stdin*, you may want to redirect *stdin* to a file. Also, depending on the setting of `tostop` by the **stty** command, output from a background job can cause it to be suspended.

Moving a Job to the Background

Suppose you want to move a foreground job to the background, where it can run while you enter other commands in the foreground. To put a job in the background:

1. Stop the job by entering <EscChar-Z>.
2. Enter the **bg** command. You may need to specify the job identifier with **bg** if there is more than one 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 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).

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.

For example, here the **ps** command displays the status of started processes:

```
    PID TTY       TIME COMMAND
 262148 tty       2:46 /bin/sh
 196614 tty       0:22 ./myprog
  65543 tty       0:13 /bin/grep
 196616 tty       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.

TIME The amount of central processor time the process has used after 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. Read the command description of **ps** in *z/VM: OpenExtensions Commands 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

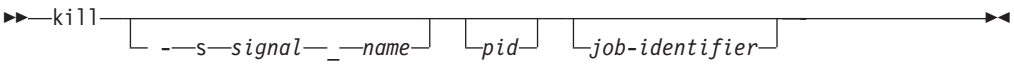
To cancel a foreground job, enter <EscChar-C>. The command stops and the shell displays the shell prompt.

Canceling a Background Job

To cancel a background job, use the **kill** command.

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 is:



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 every process in a 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:

  ```
  kill -s KILL %7
  ```

  The **-s** option specifies the signal by name.
- 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:

  ```
  kill -s -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:

```
kill -s -STOP %3
```

stops the background job with the job identifier 3.

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.

Running a Job in the Background after Exiting

If you want to start a long-running command and have it keep 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.

To exit the shell in this situation, enter the **exit** command. **nohup** ensures that the command does not end when the creating process ends. No applications survive a logoff or restart of CMS.

The **nohup** command is most practical when issued from a subshell. If you enter **nohup** from the login shell, the shell will end and the background command will continue. CMS, however, will not read commands from the command line until the background command ends. Other activity in the CMS session, such as active windows under the CMS desktop will continue to be active.

Chapter 8. Running OpenExtensions Applications

Because an OpenExtensions application is simply a program that uses POSIX services, it need not be distinguished from any other CMS application. Users can run OpenExtensions applications without entering the shell. How the application is invoked depends upon whether it resides in the CMS record file system or in the byte file system (BFS).

If the OpenExtensions application resides on an accessed minidisk or SFS directory, it has a CMS file ID in which the file name is the name of the application and the file type is MODULE. It is run, like any other CMS module file, by entering its name at the `Ready;` prompt. If the application resides in BFS, it must be run by using the **OPENVM RUN** command at the `Ready;` prompt. BFS directories are not searched during CMS command resolution, so no BFS files will ever be invoked as a result of entering a command at the CMS command line. In addition, applications in BFS have directory paths associated with them and names longer than the eight characters of a CMS command name.

If you want to run the file `application1.a` in the directory `/u/home/apps`, you would issue:

```
OPENVM RUN /u/home/apps/application1.a
```

CMS keeps track of the current working directory, so you could first change the working directory to the directory that contains your application and then avoid entering it on the **OPENVM RUN** command. For example,

```
OPENVM SET DIRECTORY /u/home/apps
OPENVM RUN application1.a
```

When running OpenExtensions programs, you should be aware of the following differences between how programs are started in the shell environment compared with how they are started directly from CMS:

- The shell has mechanisms for setting environment variables that can be interrogated by applications. In CMS, the principle way to set environment variables is to set them in the CENV group of GLOBALV. The C/C++ OpenExtensions application will initialize its environment variables from this GLOBALV group.

  If the OpenExtensions application is started through the **OPENVM RUN** command, the environment variables *HOME*, *LOGNAME*, *PATH*, and *SHELL* are initialized even if they are not defined in GLOBALV. *LOGNAME* is set to the lowercase representation of the z/VM user ID. *HOME*, *PATH*, and *SHELL* default to `/`, `/bin`, and `/bin/sh`, respectively.
- If an OpenExtensions application resides on an accessed minidisk or SFS directory, it can be invoked by name from the CMS command line. However, because it does not reside in BFS, it has no BFS permission settings. This means that all such OpenExtensions applications are executable, as if they had the *execute* permission set.
- To start an application in a strictly-conforming POSIX environment, you must start from the shell or by means of the **OPENVM RUN** command. An application started from the CMS command line is not guaranteed to have the environment variables *HOME*, *LOGNAME*, and *PATH* appropriately set. The user is responsible for giving values for these variables if the application has dependencies on them.

- An external link can be created in BFS that points to a CMS module in the record file system. This program can be run through the **OPENVM RUN** command, but care must be taken to assure that programs set up to be so invoked are capable of handling the POSIX entry conditions as defined by the **exec()** function.
- Applications started by using **OPENVM RUN** or the shell are automatically given the run-time option **POSIX(ON)**. Applications invoked from the command line must be given the **POSIX(ON)** option explicitly. This is done either by coding a `#pragma runopts(POSIX(ON))` statement in the C/C++ application, or by passing the run-time option at invocation time. This is done by specifying the option after the command name and followed by a slash (/). The parameters passed to the command follow the slash. For example, to invoke the program `myappl` and pass it two parameters, `parm1` and `parm2`, the user would enter the following at the command line:

  ```
  myappl posix(on) / parm1 parm2
  ```

  Multiple run-time options can be specified. To invoke the program mentioned above and pass it an environment variable in addition to those defined by GLOBALV, the user would enter:

  ```
  myappl env('OUTDIR=/tmp/out1') posix(on) / parm1 parm2
  ```

Chapter 9. Communicating with Other Users

Within the shell, you can send and receive messages using the **mailx** command. 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.

You can also use CMS facilities to communicate with other users, including those on remote systems. The **TELL** command sends short messages to a user or set of users. The **NOTE** command puts the user in an editing session to compose a longer message. This note is then sent to one or more users. The note is subsequently read by using the **PEEK** command to view the note file in the reader.

The advantage of using the CMS **TELL** and **NOTE** commands is that they allow communication with all CMS users, not just those using the OpenExtensions shell. **mailx** cannot be used to send messages to users on other VM systems.

Sending Messages

You can send a message using:

- The **mailx** command to communicate in the shell. If you use **mailx** to send a message, your correspondent must use **mailx** to receive it.
- CMS commands **NOTE**, **TELL** or **MSG** to communicate with any CMS user (including OpenExtensions users). If you use CMS to send a mail message, your correspondent must use CMS to receive it.

To Another User

If you use the shell frequently, you can use the **mailx** shell command.

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/.mailrc`. 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/.mailrc` file that sets these variables for your personal use. These variables are described in *z/VM: OpenExtensions Commands 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.

You can send a message to one or more users at a time. The following example is a message sent to several users. The words in italics are output from **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
~.
EOT
```

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.

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 OpenExtensions user you want to receive the message. The simplest address is the CMS 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 command:

```
alias test pfunt lulu detsch naga
```

at the **mailx** prompt (because this is a **mailx** command, not a shell command). After you do that, when you send a message to the address alias `test`, it will go to all the login names you specified.

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 `$HOME/.mailrc`.

To A VM Operator

To send a message to the VM operator, you should use the CMS **TELL** command. VM recognizes the user ID of `OP` as the system operator. For example, to send a message to the operator enter:

```
tell op are the tapes ready yet?
```

Recall that the **TELL** command is a CMS command, so to enter the command from the shell you will need to prefix it with **cms**.

Receiving Messages 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 displaying new messages and deleting old ones. 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 brian             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 some help information. 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 `brian` and has the subject 'softball'. It was sent on July 15 at 4:17 p.m., and contains 6 lines and 95 characters.

The question mark (?) is the **mailx** program prompt; it indicates that you can enter **mailx** subcommands now. To read the first message, try the subcommand **1**, and to read the subsequent message, try the subcommand **n** (next message):

```
? 1
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.
?
? n
Message  2:
From homebrw Thu Jul  15 15:03
To: smitha
Subject:  lunch plans

Have lunch plans for tomorrow?  Want to get pizza?
?
```

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 **mailx** 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 **mailx** and can enter shell commands again.

For more information on **mailx**, see *z/VM: OpenExtensions Commands Reference*.

Part 3. The File System

Chapter 10. An Introduction to the Byte File System

OpenExtensions 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 *BFS file space*. Typically, a user has all or part of a BFS file space mounted as the *root directory*.

VM views an entire BFS file space as a **Byte File System**. Each Byte File System is a mountable file system.

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 byte file system is called a *BFS file*. BFS files are byte-oriented, rather than record-oriented, like CMS record files on minidisks or in the Shared File System (SFS). You can copy BFS files into CMS record files, and you can copy CMS record files into the Byte File System.

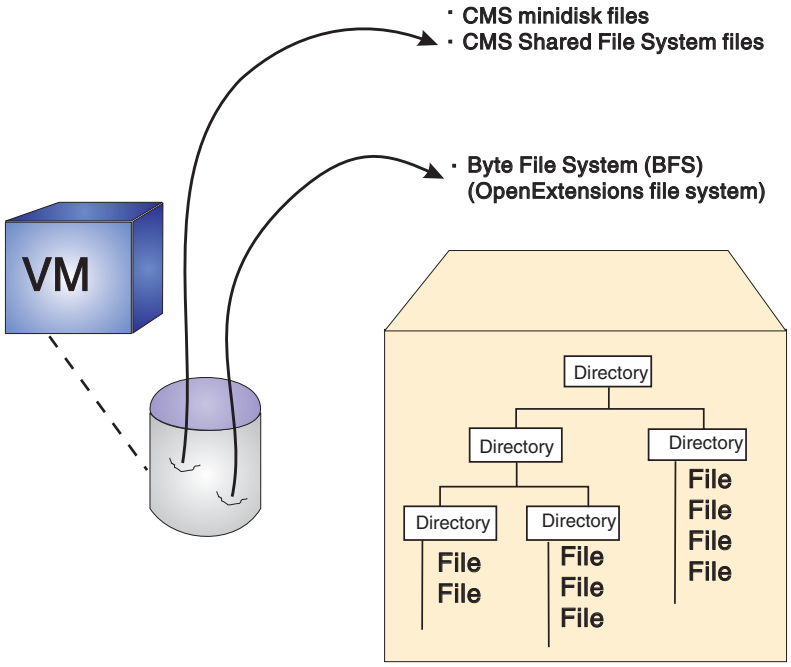


Figure 3. The Byte File System

The OpenExtensions shell 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.

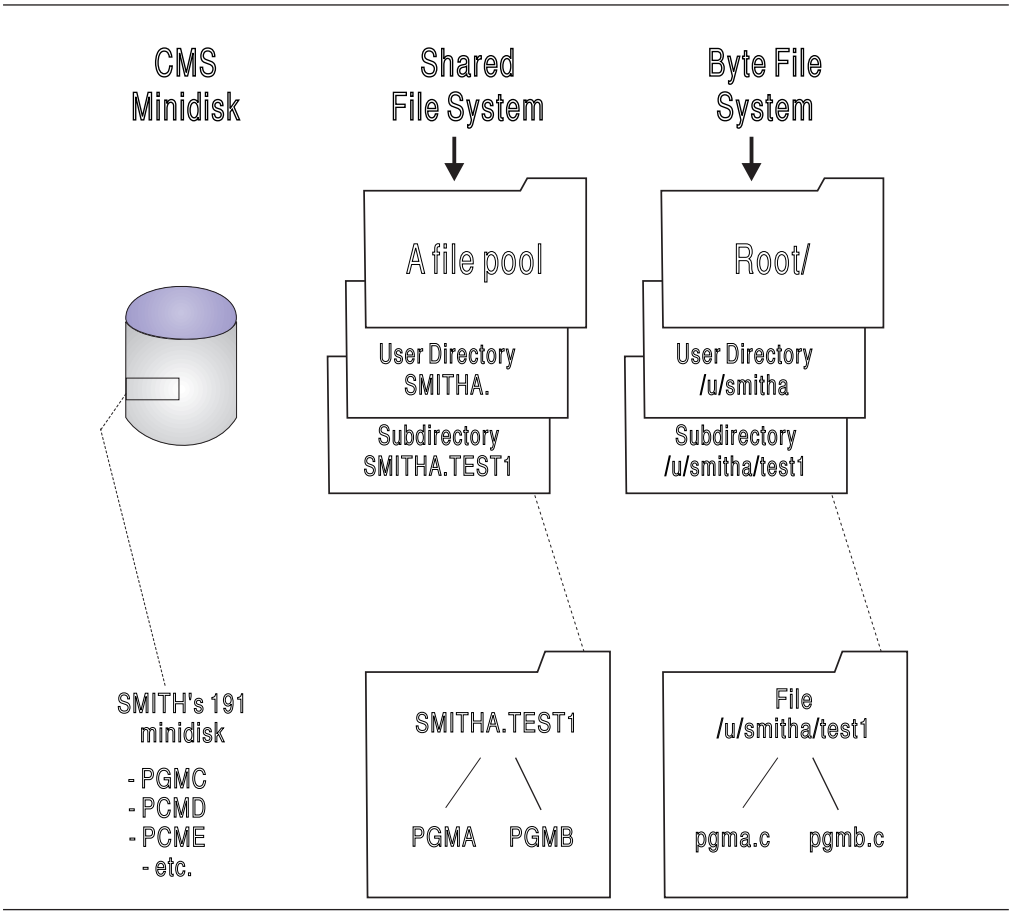


Figure 4. Comparison of CMS Record Files and the Byte File System

In Figure 4, you see that:

- The user's 191 disk or SFS file space is analogous to a user directory (`/u/smitha`) in the byte file system. Typically, one user controls all the files which reside on his 191 minidisk or within his SFS file space. For example, the files belonging to the CMS user ID SMITHA all reside in file space SMITHA. There could be a file C MODULE in SMITHA.TEST1 and SMITHA.TEST2 and a LIST EXEC file in SMITHA.TEST1 and SMITHA.TEST2.

  In the byte file system, SMITHA would have a user directory named `/u/smitha`; under that directory there could be directories named `/u/smitha/test1` and `/u/smitha/test2`.

  In z/VM, SMITHA could also have his own byte file system. Whether you have your own byte file system or your own user directory is a decision made by your system programmer.
- An SFS file space is most akin to a user directory in the byte file system. In an SFS directory such as SMITHA in the VMSYSU file pool, you could have subdirectories PGMA, PGMB, and so on—for example, SMITHA.TEST1 can have a PGMA MODULE and a PGMB MODULE. 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 byte file system can be read by all programs as soon as it is written. Data is written to the byte file system when a program issues an **fsync()**.

Before learning about byte file system capabilities, you need to understand these concepts:

- The root file system and mountable byte file systems
- Directories
- Files
- Path and path name

The Root File System and Mountable Byte File Systems

The system programmer (SFS administrator) defines one or more Byte File Systems; subsequently, a user can mount other byte file systems or pieces of byte file systems or remote file systems on directories within the file hierarchy. Altogether, the root file system and mountable file systems comprise the user's view of the Byte File System used by OPENVM commands.

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 5, the slash (/) symbol at the top represents the *root directory,* from which all other directories are descended. There are seven directories branching from the root. Each of these directories, in turn, has its own system of subdirectories and files. For example, `tty` is a file in the directory `/dev` and `localedef` is a subdirectory in the directory `/usr/lib/nls`.

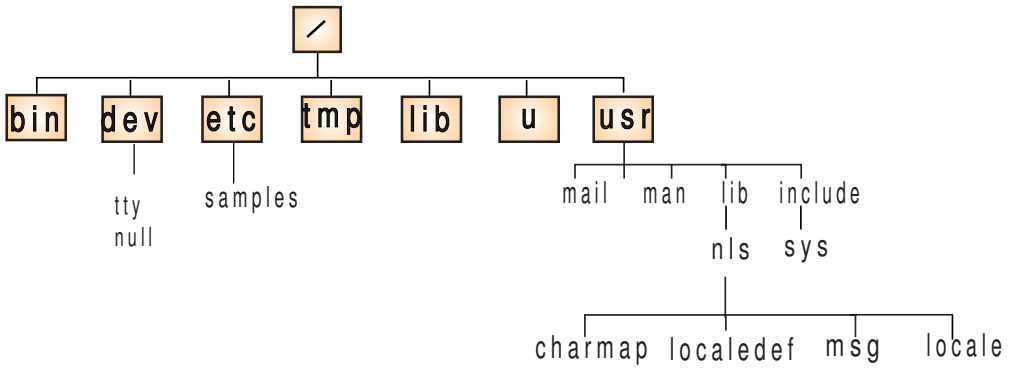


Figure 5. Organization of the Byte File System

In the BFS, such I/O destinations as terminals, sockets, and named pipes are defined as files. Those are called *special files*.

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 OpenExtensions 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 BFS, in addition to directories:

- A **regular file** is an identifiable (named) unit of text or binary data information. A file can be C/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, or 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** is a file that defines
  - A terminal (for example, `/dev/tty`).
  - A null file (for example, `/dev/null`).
  - A file descriptor file (for example, `/dev/fd`*n*).
  - A path name that represents a UNIX domain socket. The path name is assigned by the application programmer; there is no convention for the name. The socket library creates the file when a server program readies itself to accept client connections.
- A **FIFO special file** is a file typically used to send data from one process to another 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 points to some other BFS file. It permits users and programs to refer to that other file by either the other file's own path name or the path name of the symbolic link. You can create a symbolic link to a file or a directory.
- An **external link** can be thought of as a type of symbolic link: a link to an object outside of the byte file system. Typically, it contains information about how to access a CMS record file.

Path names representing special files can be used anywhere path names for regular files can be used, such as redirection constructs or in the syntax of a command that requires a path name.

Files Not in the BFS

There are two types of unnamed files that do not exist in the BFS:

- An **unnamed pipe.** 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 byte file system, and it vanishes when the last process using it closes it.

  A program creates a pipe with the **pipe()** function.
- A **socket**. 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.

  A program creates a socket with the **socket()** function.

Path and Path Name

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 *path name*. Path names are used as arguments for commands.

An *absolute path name* 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 file name. The search for the file begins at the root and continues through the elements in the path name until it gets to the final name. For example: `/u/smitha/projectb/plans/1dft` is the absolute path name for `1dft`, the first draft of the plans for a particular project that a user named Alice Smith (SMITHA) is working on.

A *fully-qualified path name* is a sequence that begins with the special keyword string **/../vmbfs:**, a file pool identifier, and a byte file system identifier. This special path name format allows the use of BFS files *without mounting* the byte file system. An example of a fully-qualified path name for file PGMA is: `/../vmbfs:vmsysu:bfs/u/smitha/pgma`.

A fully-qualified Network File System (NFS) path name is a sequence that begins with the special keyword string /../nfs, a foreign host name followed by a slash, then the name of the remote directory. This special path name format cannot be used without mounting the file system.

Instead of using the absolute or fully-qualified path name with shell and OPENVM commands, you can specify a path name as relative to the working directory; this is called the *relative path name*. In most cases, a user can specify a particular file without having to use its absolute path name. A relative path name 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 path name for the file `/u/smitha/projectb/plans/1dft` as `plans/1dft`.

A path name can be up to 1023 characters long, including all directory names, file names, and separating slashes. For path names and file names, use characters from the POSIX portable character set.

The system performs *path name resolution* to resolve a path name to a particular file in a file hierarchy. The system searches from element to element in a path name in order to find the file.

Requirement for a Fully-Qualified Path Name

In one situation, a fully-qualified path name is required. The **OPENVM MOUNT** command requires that you use the fully-qualified form of the BFS, BFS subdirectory, or NFS path name.

You may also specify a fully-qualified NFS path name on OPENVM UNMOUNT.

Resolving a Symbolic Link in a Path Name

A *symbolic link* is a file that contains the path name for another file; that path name can be relative or absolute. If a symbolic link contains a relative path name, the path name is relative to the directory containing the symbolic link.

If you use a symbolic link as a component of a path name, during path name resolution the original path name is changed. How it changes depends on whether the symbolic link contains a relative or absolute path name. For example, consider the path name `/u/turbo/dlg/lev1`:

- If `dlg` is a symbolic link containing the relative path name `dbopt/pgma/src`, `dlg` is replaced by the relative path name. This is how it resolves:

  `/u/turbo/dlg/lev1` → `/u/turbo/dbopt/pgma/src/lev1`.

- If `dlg` is a symbolic link containing the absolute path name `/usr/bin/dbopt/pgma/src`, then the components in the original path name that preceded `dlg` are replaced by the absolute path name 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 path name.

External Links

An external link can be thought of as a type of symbolic link that refers to an object outside of the BFS. See “Creating an External Link” on page 107 for more information and *z/VM: OpenExtensions Commands Reference* for more about the **OPENVM CREATE EXTLINK** command.

Using Commands to Work with Directories and Files

You can use OPENVM commands to do certain tasks with the byte file system. Some of these are tasks that UNIX users traditionally perform while in the shell. Because these are CMS commands, you can perform these byte file system tasks whether or not you have the OpenExtensions Shell and Utilities installed.

The following table describes the OPENVM command and the equivalent shell command.

Function	Shell	CMS
Change a working directory.	**cd**	**OPENVM SET DIRECTORY**
Change the group owner of a file or directory. To use this command, you must be a superuser or the owner of the file or directory.	**chgrp**	**OPENVM OWNER**
Change access permission to a directory or file. To use this command, you must have appropriate privileges—you must have write authority, or be the file owner, or be a superuser.	**chmod**	**OPENVM PERMIT**
Change the owner or group of a file or directory. To use this command, you must be a superuser.	**chown**	**OPENVM OWNER**
Copy a file.	**cp**	**OPENVM GETBFS**
Copy a file.	**cp**	**OPENVM PUTBFS**
Create or edit text in a file.	**ed**	**XEDIT**
Link another name to a file (in addition to its original name).	**ln**	**OPENVM CREATE LINK**
List the files in a directory.	**ls**	**OPENVM LISTFILE**
Create a directory. The mkdir command has an option for creating intermediate directories in a path name.	**mkdir**	**OPENVM CREATE DIRECTORY**

Function	Shell	CMS
Make a named pipe (FIFO special file).	**mkfifo**	-
Make a character special file or a named pipe (FIFO special file). To use this command to create a character special file, you must be a superuser.	**mknod**	-
Move a file from one directory to another directory, or rename a file or directory.	**mv**	**OPENVM RENAME**
Display your working directory.	**pwd**	**OPENVM QUERY DIRECTORY**
Remove (erase, or delete) a file from a directory.	**rm**	**OPENVM ERASE**
Remove (erase, or delete) a directory that is empty of files.	**rmdir**	**OPENVM ERASE**
Add a mountable byte file system or byte file system sub-directory tree to the file hierarchy.	-	**OPENVM MOUNT**
Create a byte file system path name to be used to reference a file or other data that resides outside of the byte file system.	-	**OPENVM CREATE EXTLINK**
Create a byte file system path name to be used to reference an object residing in one byte file system using a path name in the same or another byte file system.	**ln**	**OPENVM CREATE SYMLINK**
Display information associated with symbolic or external links.	-	**OPENVM QUERY LINK**
Define the file creation mask to be used when creating a BFS object.	**umask**	**OPENVM SET MASK**
Display the file creation mask values in effect.	**umask**	**OPENVM QUERY MASK**
Display what is mounted in your hierarchy.	-	**OPENVM QUERY MOUNT**
Remove a byte file system or byte file system subdirectory tree from your hierarchy.	-	**OPENVM UNMOUNT**
Execute an application that resides in the byte file system, or has an external link to a CMS module in the record file system.	Just type in the name of the application at the shell prompt.	**OPENVM RUN**
Process archive tapes.	-	**OPENVM PARCHIVE**

Where You Can Enter a CMS Command

For example, you can enter a CMS command:

- At the CMS `Ready;` prompt.
- At the XEDIT `====>` prompt.
- In the shell, by prefixing the command with **cms**.

CMS maintains a working directory, primarily to support execution of OpenExtensions applications outside the shell. If you specify a relative path name on one of the OPENVM commands provided by CMS, or on XEDIT, the working directory as set by **OPENVM SET DIRECTORY** is used to construct the full directory name.

Locking

Locking is coordinated across both local and remote users. No locking is done for objects in NFS-mounted file systems.

External Links

An external link can be thought of as a symbolic link to non-byte file system files. It associates a CMS record file with a BFS path name. The external link lets you transparently access a CMS record file using a POSIX-conforming path name. For example, you can create an external link to map `/u/smitha/pgma` to PGMA MODULE A for the CMS record file. See *z/VM: OpenExtensions Commands Reference* for more information about **OPENVM CREATE EXTLINK**.

Security for the File System

No additional security product is necessary for the byte file system. Security is ensured as part of the base product.

Chapter 11. 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

CMS and the shell always identify 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 file name with a command. If you want to work with a file in another directory, you can change your working directory, using **OPENVM SET DIRECTORY** or the **cd** shell command and naming the new directory. (Instead of changing directories, you could use relative notation or fully-qualified notation to access a file in a different directory; see "Using Notations for Relative Path Names" on page 96 for more information.)

When you type the **OPENVM SHELL** 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 **OPENVM QUERY DIRECTORY** command, or the **pwd** shell 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 is she used the **pwd** command:

```
/u/smitha
```

`/u/smitha` is the *pathname* of her working directory.

If Alice Smith enters:

```
OPENVM SET DIRECTORY projecta
   or
cd projecta
```

the `projecta` subdirectory of her home directory becomes her working directory. If she enters:

```
OPENVM QUERY DIRECTORY
   or
pwd
```

it displays (for **pwd**): `/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 **OPENVM SET DIRECTORY** or the **cd** shell 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 byte file system by using one of these commands and the path name for the directory:

```
OPENVM SET DIRECTORY pathname
   or
cd pathname
```

See Chapter 13 for more information on directory permissions.

When you want to go to your home directory, just enter the **cd** shell command with no arguments:

```
cd
```

To change to a directory other than your home directory, you must supply the path name. For example, if Alice Smith is working in her home directory (`/u/smitha`) and she wants to switch to her `projectb` directory, she enters:

```
OPENVM SET DIRECTORY projectb
   or
cd projectb
```

To check that she has changed directories, Alice enters:

```
OPENVM QUERY DIRECTORY
   or
pwd
```

and the system displays (for **pwd**): `/u/smitha/projectb`.

Using Notations for Relative Path Names

To change directories quickly or to work with a file name in another directory, use these relative path name notations:

dot notation (**.** and **..**)
tilde notation (**~**)

Dot Notation

A convention used in the shell environment is to use **.** (dot) and **..** (dot dot) to represent certain directories.

. (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 byte file system structure.

If one of these is used as the first element in a relative path name, 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 OpenExtensions 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 (if `dir` is an existing 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. However, the OpenExtensions shell does not have access to variables in other user's virtual machines, so any login name including the issuer's will resolve to **/** (a slash). **Note:** In the OpenExtensions shell, your *login name* is your VM user ID.

Example

For example, suppose you are working in the shell 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 /u/turbo/appl/hdr/limits
```

and if you are logged on as `turbo`, you could also use:

```
cat ~/appl/hdr/limits
```

Creating a Directory

Using CMS

To create a new directory, enter:

```
OPENVM CRE DIR directory_name
```

where *directory_name* specifies the path name of the directory to be created. The path name can be an absolute path name, a relative path name, or a fully-qualified path name. Specify the name, which can be up to 1023 characters long, in single quotation marks or double quotation marks when any of the following characters are part of the name:

A blank space

(Left parenthesis

) Right parenthesis

‘ A single quotation mark

“ A double quotation mark

***** Asterisk

if it contains blanks or other special characters. The **OPENVM SET MASK** command can be used before the **OPENVM CREATE DIRECTORY** command to mask off permissions when it is created, or the **OPENVM PERMIT** command can be used after the **OPENVM CREATE DIR** command to change them after the directory is created.

If user Alice Smith is in her current working directory `/u/smitha` and wants to create a directory *umods* using a relative path name, she would enter:

```
OPENVM CRE DIR umods
```

The directory `umods` is one level below her working directory `smitha`; its full path name is `/u/smitha/umods`.

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=r-x
other=r-x

Here *execute* permission means permission to search the directory. The octal representation of these permissions is `755` (7 for the owner permission bits; 5 for the group permission bits; 5 for the other permission bits).

The new directory, `projecta`, is one level below her working directory. Figure 6 on page 99 shows this relationship. If you do not specify an absolute path name 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.

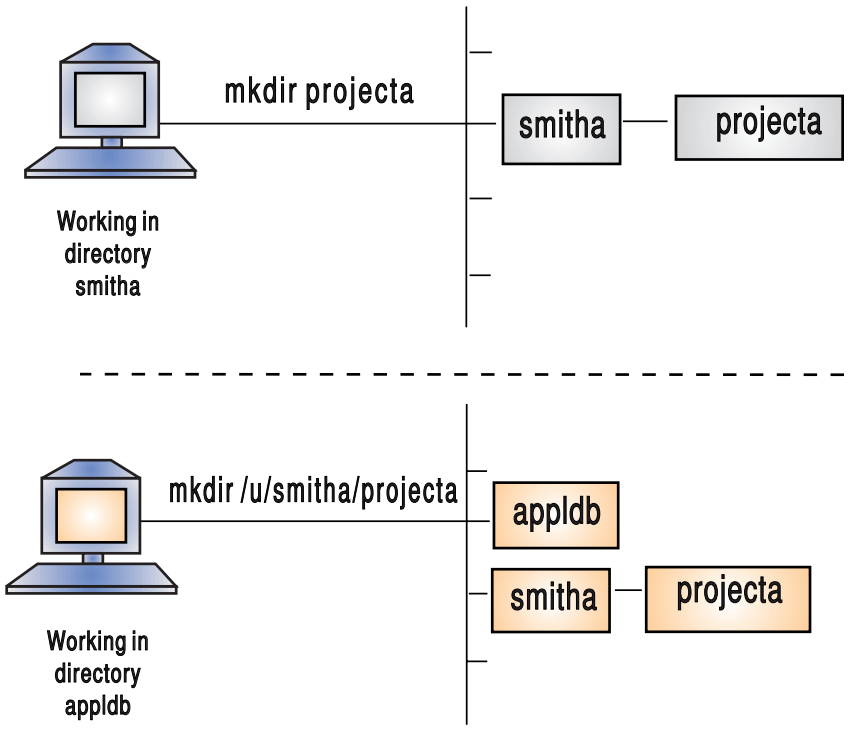


Figure 6. Creating a New Directory

If you want to create a new directory that is *not* under your working directory, specify an absolute path name or a fully-qualified path name. Both directory names and file names can be up to 255 characters long. You may want to adopt some naming convention that lets you distinguish between directory names and file names.

Your business may have adopted naming conventions for directories. For example, a typical convention is for each user to be assigned a directory based on the VM 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 13.

Removing a Directory

Using CMS

You can remove an empty directory (one with no files or subdirectories) from the byte file system with the **OPENVM ERASE** command. The format of the command is:

```
OPENVM ERASE directory
```

Using the Shell

You can remove an empty directory (one with no files or subdirectories) from the byte 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.

Attention:

This will remove the entire subtree.

To delete the files in a directory and the directory itself in one step from the shell, 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. You should also use the **-i** option so that you will be prompted to confirm the deletions:

```
rm -ri file
```

Listing Directory Contents

Using CMS

The **OPENVM LISTFILE** command lists the contents of a directory. To see the contents of your working directory, enter:

```
OPENVM LIS
```

To use the **(SUBDIRECTORY** option to see the contents of directory `/sam` and all of its subdirectories, enter:

```
OPENVM LIS /SAM  (SUBD
```

You can request different types of information from the **OPENVM LISTFILE** command. Using different options, you can ask for information about byte file system attributes, names, and owners.

- Attributes

  To display attribute information about objects in your current working directory and the directories under it, enter:

  ```
  OPENVM LIS (SUBDIR
     or
  OPENVM LIS (ATTR SUBDIR
  ```

  You will see a display that includes path names, update times, and other statistics like this one:

  ```
  Directory = '/'
  Update-Dt  Update-Tm Type  Links   Bytes Path name component
  02/02/1995 12:23:55   D      -       -   'International_data'
  02/02/1995 12:22:55   D      -       -   'US_cities_and_towns'
  02/15/1995 14:32:35   F      1     20956 'US_cities_and_towns/Anaheim_File'
  02/15/1995 14:32:35   F      1      2346 'US_cities_and_towns/Boston_File'
  02/15/1995 14:32:35   F      2     10956 'US_cities_and_towns/Detroit_File'
  02/15/1995 14:32:35   F      1     34556 'US_cities_and_towns/Endicott_File'
  ```

- Names

  To display information about names, enter:

  ```
  OPENVM LIS (SUBD NAMES
  ```

  You will see a display that includes a special system-generated file name and file type. If you are an SFS administrator, this allows you to use BFS files using some CMS record file system commands. (See *z/VM: CMS Commands and Utilities Reference* for more information.) Here is an example of what you might see:

  ```
  Directory = '/'
  File name File type Byte file system  Type  Path name component
  3         0         SERV1009:BFS1.     D    'International_data'
  2         0         SERV1009:BFS1.     D    'US_cities_and_towns'
  ```

```
4         0         SERV1009:BFS1.     F    'US_cities_and_towns/Anaheim_File'
5         0         SERV1009:BFS1.     F    'US_cities_and_towns/Boston_File'
6         0         SERV1009:BFS1.     F    'US_cities_and_towns/Detroit_File'
7         0         SERV1009:BFS1.     F    'US_cities_and_towns/Endicott_File'
Ready;
```

```
Directory ='bfsdname'
Filename Filetype Byte File System   Type  Path name Component
  fn        ft    bfsid               t    pname
  .         .     .                   .    .
  .         .     .                   .    .
```

- Owners

  To display information about owners, enter:

  ```
  OPENVM LIS /MYDIR  (OWNERS
  ```

  You will see a display in this format:

  ```
  Directory ='bfsdname'
  POSIX UID   POSIX GID  Permissions Type  Path name Component
   uid          gid      rwx rwx rwx  t    pname
    .            .           .        .     .
    .            .           .        .     .
    .            .           .        .     .
  ```

Using the Shell

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 examine, 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 file names 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
=	Identifies a socket file

If there is no character following the file name, 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.

Comparing Directory Contents

Using the Shell

You can use the shell 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

Using the Shell

The shell command **find** searches a directory and lists the names of all the files having 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 file name 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 many directories and subdirectories. For more information on the **find** command, see *z/VM: OpenExtensions Commands Reference*.

Chapter 12. Working with Files

This chapter covers the topics:
- Using an editor to create a file
- Naming files
- Deleting a file
- 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

Using an Editor to Create a File

When working at the host system, you have three editors to use to create and change files:

- XEDIT, a full-screen editor that you can call from CMS or the shell
- The **ed** editor, a line editor that you can call from the shell or CMS
- The **sed** stream editor, a noninteractive editor. It is intended for *systematic* editing; you call the editor with a file of editing commands and a target data file and it produces an edited target file, with no user interaction.

For details about the editors, see Chapter 14.

You can control access to your directories and files. When you first create a directory or file, *access permissions* are set for them. You can change these permissions whenever you want. See Chapter 13 for more information on access permissions.

Naming Files

A file name can be up to 255 characters long. To be portable, the file name should use only the characters in the POSIX portable file name 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 file name.

Doublebyte characters are not supported in a file name and are treated as singlebyte data. *Using doublebyte characters in a file name 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 path name.

The OPENVM commands and the OpenExtensions shell are case-sensitive and distinguish characters as either uppercase or lowercase. Therefore, `FILE1` is not the same as `file1`.

A file name can include one or more suffixes, or *extensions*, that indicate 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 file name `dbmod3.c`. Having groups of files with identical suffixes makes it easier to run commands against many files at the same time.

Note: If you want to use **c89** or **cxx** to compile and build, C source file names *must* have a `.c` suffix. C++ source file names *must* have a `.cpp` or `.cxx` suffix.

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:

The shell Case-sensitive. In the file system, you can use mixed-case file names.

CMS Case-sensitive. Follow the syntax rules for the OPENVM command. For instance, make sure to enclose file names in quotation marks when using names containing blanks, left parentheses, and other special characters on the OPENVM commands.

Deleting a File

Using CMS

The command **OPENVM ERASE** can delete a file. For example,

```
OPENVM ERASE file1
```

erases `file1` in your current working directory.

Using the Shell

The command **rm** can delete, or "remove", several files at the same time.

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.

Identifying a File by Its Inode Number

In addition to its file name, each file in a file system has a unique identification number called an *inode number*. The inode number refers to the physical file, the data stored in a particular location.

A directory entry joins a file name with the inode number that represents the physical file.

Using CMS

To display the inode numbers of the files in a directory, use the **OPENVM LISTFILE** command with the **(NAMES** option. The inode number is the number shown in the *Filename* column.

Using the Shell

To display the inode numbers of the files in your working directory, just enter:

```
ls -i
```

If Alice Smith enters that command for her `projecta` 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 path name, 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 path name 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 path name for a file that has a long path name.

A file can have an unlimited number of links to it.

Using CMS

You can use the **OPENVM CREATE LINK** command to create a hard link or the **OPENVM CREATE SYMLINK** command to create a symbolic link.

Using the Shell

You can use the **ln** command to create a hard link or a symbolic link.

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:

Using CMS

```
OPENVM CRE LINK old new
```

Using the Shell

```
ln old new
```

Thus, *new* is the new path name for the existing file *old*. In Figure 7, `/u/benson/projecta` is the new path name for the existing file `/u/smitha/projecta`.

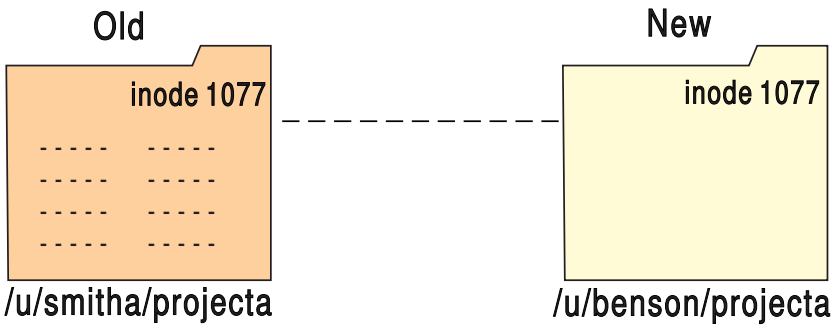


Figure 7. A Hard Link: A New Name for an Existing File. The hard link has an identical inode number.

When you create a hard link to a file, the new file name shares the inode number of the original physical file, as shown in Figure 7. 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

A *symbolic link* is another file that contains the path name for the original file—in essence, a reference to the file. 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 can refer to a path name for a file that does not exist.

To create a symbolic link to a file, use this command format:

Using CMS

```
OPENVM CRE SYM old new
```

Using the Shell

```
ln -s old new
```

Thus *new* is the name of the new file containing the reference to the file named *old*. In Figure 8 on page 107, `/u/benson/projecta` is the name of the new file that contains the reference to `/u/smitha/projecta`.

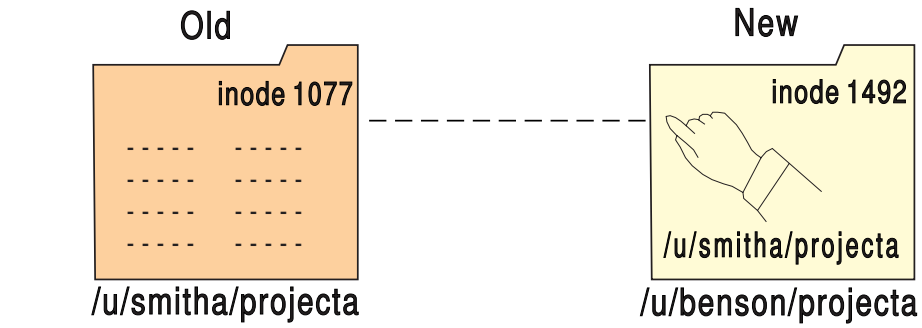


Figure 8. A Symbolic Link: A New File. A symbolic link has its own inode number.

When you create a symbolic link, you create a new physical file with its own inode number, as shown in Figure 8. Because a symbolic link refers to a file by its path name 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 path name is handled during path name resolution, see “Resolving a Symbolic Link in a Path Name” on page 91.

Creating an External Link

An *external link* is special type of link; it is a a file that contains the name of an object kept outside of the byte file system. Using an external link, you can associate that object with a POSIX-conforming BFS path name. If the link is to a CMS record file, you can use the path name to access the file.

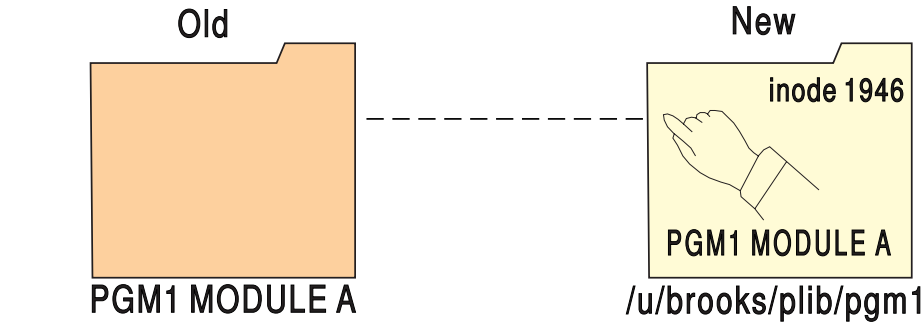


Figure 9. An External Link: A New File. An external link has an inode number.

To create an external link to a CMS record file, use this command format:

```
OPENVM CREATE EXTLINK path type file_id
```

In Figure 9, `/u/brooks/plib/pgm1` is the name of the new file that contains the reference to the CMS record file PGM1 MODULE on user BROOKS' A-disk.

Limitations of an External Link:

1. You must use **OPENVM CREATE EXTLINK** to create it.
2. In order to use the external link, the file pointed to must be on an accessed minidisk or accessed SFS directory.

Deleting Links

Using CMS

To delete a file with hard links, you must use **OPENVM ERASE** for every name for the file. The contents of the file do not disappear until you remove the last link.

To delete symbolic links or external links, use **OPENVM ERASE**.

Using the Shell

To delete a file with hard links, you must use **rm** against every name for the file. The contents of the file do not disappear until you remove the last link.

To delete a file with symbolic links, you use **rm** against its original name. 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 an external link, use **rm** against the name of the external link.

If you delete a CMS file that is externally linked, the remaining external link refers to a file that no longer exists.

Renaming or Moving a File or Directory

Using CMS

You can use the **OPENVM RENAME** command to move or rename files. For example,

```
OPENVM REN file1 file2
```

changes `file1` in the current directory to have name `file2`.

Using the Shell

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
```

moves the contents of `directorya` to directory `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.

Using CMS

The CMS Pipelines command has support for BFS files. The BFS stage can be used to read from BFS files, intermediate pipelines stages can be used to compare the files, and an output BFS stage can write the results to a new or existing BFS file. See *z/VM: CMS Pipelines Reference* for more information on the CMS Pipelines command.

Using the Shell

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** shell 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.text newhist.text
```

You use an 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.text newhist.text
```

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.text`, contained the line `Susan B. Anthony awoke one morning` where the second file, `newhist.text`, 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 *z/VM: OpenExtensions Commands Reference* for more details.

diff helps you determine what has changed in the time that elapsed between saving the two files. It is useful any time you have two different versions of the same file and want to check the differences.

Sorting File Contents

Using CMS

The CMS Pipelines command has support for BFS files. The BFS stage can be used to read from BFS files, intermediate pipelines stages can be used to compare the files, and an output BFS stage can write the results to a new or existing BFS file. See *z/VM: CMS Pipelines Reference* for more information on the CMS Pipelines command.

Using the Shell

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** shell 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 a character that does not itself appear in the fields as data. In the sample file `comics.lst` in `/etc/samples` (shown in Figure 10), colons are used.

```
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
```

Figure 10. A Sample File: comics.lst (Part 1 of 4)

```
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
```

Figure 10. A Sample File: comics.lst (Part 2 of 4)

```
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
```

Figure 10. A Sample File: comics.lst (Part 3 of 4)

```
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 10. A Sample File: comics.lst (Part 4 of 4)

To sort a file such as our comic book file, enter:

```
sort /etc/samples/comics.lst
```

This command sorts the list and displays it. To save the sorted list in a file, enter:

```
sort /etc/samples/comics.lst >filename
```

where *filename* is the name of the file where you want to store the sorted list. For example:

```
sort /etc/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 will usually make the output file name different from the (unsorted) input file name. If you really want to overwrite a file with its sorted contents, see the description of the **–o** flag in **sort** in *z/VM: OpenExtensions Commands Reference*.

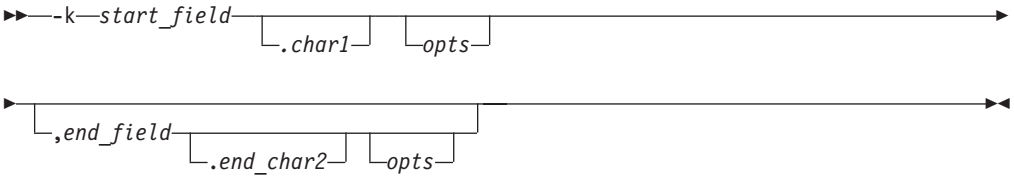
Using Sorting Keys — An Example

By default, **sort** sorts according to all the information in the record, in the order given in the record. Because 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:



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.
- *end_field* indicates which field in the input record contains the end of the sorting key. If *end_field* is not specified then the sorting key extends from the starting position to the end of the record.
- *char2* indicates which character in that field is the last character of the key. Omitting *char2* means that the key ends with the last character of the end field.

In our example, the first sorting key (referring to the year) has a *start_field* value of `4` (because the year is field 4). *char1* does not need to be specified, because the start key begins 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,4
```

The second sorting key in the example refers to the month field (3). This key has the form:

```
-k 3M,3
```

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,4 -k3M,3  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 it is usually put at the end.

Counting Lines, Words, and Bytes in a File

Using CMS

Use the **OPENVM LISTFILE** command to find out how many files are in a directory and to find out attributes for files, such as the number of bytes in the file. CMS Pipelines can also be used to count words, characters, and lines that pass through a pipeline. See the **COUNT** stage command in *z/VM: CMS Pipelines Reference*

Using the Shell

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 -l
```

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

Using CMS

The CMS Pipelines command has support for BFS files. The BFS stage can be used to read from BFS files, intermediate pipelines stages can be used to compare the files, and an output BFS stage can write the results to a new or existing BFS file. See *z/VM: CMS Pipelines Reference* for more information on the CMS Pipelines command.

Using the Shell

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 a file containing the records you wish to search. **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 quotation marks (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

So far the examples of **grep** have displayed the records that contain the desired string anywhere in the record. 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 File Names” on page 45. 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 file name, this example uses the special character ***** to represent any file name 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 *metacharacters*. Some metacharacters 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). Because 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

The OpenExtensions shell accepts much more complex patterns than the ones discussed here. The formal name for a pattern is a *regular expression*. For further information, see the appendix on regular expressions in *z/VM: OpenExtensions Commands 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 a BFS file using shell commands. With shell commands, you have the choice of browsing the file in an unformatted or formatted display.

Browsing Files Without Formatting

The OpenExtensions 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.

`tail` *`filename`* Displays the last 10 lines of the given file.

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 the **cat** command.

Browsing Files with Formatting

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 file name, 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 path name, 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. In a program, you can use byte-range locking to avoid this problem. When a file has been accessed by more than one user simultaneously, the last changes saved overwrite any previous changes. For more information about locking byte-ranges in a program, see the fcntl() function in the *XL C/C++ for z/VM: Runtime Library Reference*.

The CMS XEDIT command obtains a byte-range lock for the entire file when reading it into the XEDIT session and when writing it to the byte file system.

Backing Up and Restoring Files: The Options

For someone who has SFS administrator authority there are system-level commands available. These commands are **FILEPOOL BACKUP**, **FILEPOOL FILELOAD**, **FILEPOOL RESTORE** and **FILEPOOL UNLOAD**. With these commands you can backup a file space or storage group and you can restore either a file space, a group or an individual file. For more information, see *z/VM: CMS File Pool Planning, Administration, and Operation*.

There are three commands you can use to back up and restore files and directories: **pax**, **cpio**, and **tar**. The **pax** command combines the power of the two popular commands **cpio** and **tar**.

You use these commands to create an *archive file* that records the contents of files and directories in a specific format. A file stored inside an archive file is sometimes called a *component file*; likewise, a directory stored inside an archive file is sometimes called a *component directory*.

If you use absolute path names when creating an archive file, then the files in the archive are restored to the same position in the hierarchy, regardless of what directory you are working in when you restore the files. For example, `/tmp/c/proga.c` is a directory and you archive it with the command:

```
tar -cvf archive /tmp/c/proga.c
```

when you restore from *archive*, the restored files are placed in directory `/tmp/c/proga.c`.

On the other hand, say you were working in the directory `/tmp/data` and entered the **tar** command to write an archive file without specifying the absolute path name for the directory:

```
tar -cvf archive
```

If you happened to be in a different directory when restoring the *archive* file, say in `/tmp/usr`, all the component files would be restored in `/tmp/usr`.

Code Page Conversion: If you need to convert files from one character set to another, use the **pax** command with the **-o** option. See “Converting Between Code Pages” on page 123.

Using cpio to Back Up and Restore Files

cpio reads and writes either a compact binary format header or an ASCII format header. The **cpio** command has no limit on the length of a file name. For information on the cpio archive file format, see Appendix D.

In these examples, `/tmp/posix` is the working directory.

Backing Up a Complete Directory

Using the Shell

To back up a complete directory, including the subdirectories and their contents, into a file takes two commands. First a shell command,

```
find directory_name | cpio -o > archive_file
```

and then a CMS command:

```
OPENVM GETBFS archive_file file_id (BFSLINE 1
```

where *archive_file* is an absolute path name and *file_id* is a CMS record file specification.

The **find** command extracts path names from the specified directory. Its contents are piped to **cpio**, which creates an archive file in your working directory. The **OPENVM GETBFS** command copies the archive file into the specified CMS record

file. For example, to back up the directory /tmp/posix/testpgm into the file TESTPGM CPIO, enter these two commands:

```
find /tmp/posix/testpgm | cpio -o > testpgm.cpio
```

(switch environments and then enter)

```
OPENVM GETBFS /tmp/posix/testpgm.cpio TESTPGM CPIO (BFSLINE 1
```

For the **cpio** command, the **-o** option writes to an archive file—in this case, to testpgm.cpio.

After you have copied the archive file into an CMS record file, you can delete it from the BFS.

Restoring a Complete Directory from a VM File

The following commands would restore the directory in the previous example:

```
OPENVM PUTBFS TESTPGM CPIO A /tmp/posix/testpgm.cpio (BFSLINE NONE
```

(switch environments and then enter)

```
cpio -iud < testpgm.cpio
```

The **OPENVM PUTBFS** command copies the file containing an archive file into the specified directory in the file system. The **cpio** command restores the contents of the archive so that they can be accessed in the file system.

For the **cpio** command:

- The **-i** option reads an archive—in this case, from testpgm.cpio.
- The **-u** option overwrites any existing file or directory.
- The **-d** option creates any necessary intermediate directories.

Working with a Compressed Archive

To compress the archive file when it is created, enter the shell command:

```
find /tmp/posix/testpgm | cpio -ocz > testpgm.cpio.z
```

For the **cpio** command:

- The **-c** option creates the header in ASCII format. This is useful when the archive is made up of text files and is sent using data communication; it is also recommended for transferring data between different machines.
- The **-z** option compresses the archive. Adding the .z to the file suffix .cpio is a UNIX convention for specifying the file is compressed.

To restore the directory from the compressed archive file, enter the shell command:

```
cpio -icudz < testpgm.cpio.z
```

As you recall, the **-c** option indicates the header is in ASCII format.

Viewing the Contents of an Archive

To display a listing of the contents of the archive file, enter the shell command:

```
cpio -ictzv < testpgm.cpio.z
```

For the **cpio** command:

- The **-t** option lists the contents of the archive on standard output.
- The **-v** option gives more detailed, or “verbose”, information with the list.

If the file is not compressed, you do not use the **-z** option.

Restoring Selected Files from an Archive

To restore only selected files from the archive file enter the shell command

```
cpio -icdz file1.c file2.c tmpdir < testpgm.cpio.z
```

This command restores only `file1.c`, `file2.c`, and all files in the directory `tmpdir` from the archive `testpgm.cpio.z`. These files are restored only if they do not exist in your working directory or if the files in the working directory are older than those in the archive. This is useful for restoring the most up-to-date files, but not for replacing a file with an older version from the archive. (To copy an archive file to a target file even if the archive file is older than the target file, use the **-u** option.)

To restore all files *except* `file1` and `file2`, use the **-f** option:

```
cpio -icdzf file1 file2 < testpgm.cpio.z
```

Using tar to Back Up and Restore Files

tar reads and writes headers in either the original TAR format from UNIX systems or the USTAR format defined by the POSIX 1003.1 standard. With the TAR format, the length of the path name you can specify is 100 characters. With the USTAR format, the length of the path name you can specify is 255 characters. For information on the TAR archive file formats, see Appendix D.

During the backup or restore process, **tar** preserves link information.

If you will be putting the archive file on a tape, the *blocksize* that was used when writing the file should be used when reading the file.

In these examples, `/tmp/posix` is the working directory.

Backing Up a Complete Directory into a CMS Record File

To back up a complete directory, including the subdirectories and their contents, into a file, enter two commands. First the shell command,

```
tar -cf archive_file directory_name
```

and then the CMS command:

```
OPENVM GETBFS archive_file file_id (BFSLINE 1
```

where *archive_file* is an absolute path name and *file_id* is a CMS record file specification.

The **tar** command creates the specified archive file in your working directory. The **OPENVM GETBFS** command copies the archive file into the specified file.

For example, the following commands back up the directory `/tmp/posix/testpgm` into the file TESTPGM TAR:

```
tar -cvf testpgm.tar /tmp/posix/testpgm
```

(next enter the CMS command)

```
OPENVM GETBFS /tmp/posix/testpgm.tar TESTPGM TAR (BFSLINE 1
```

For the **tar** command:

- The **-c** option creates an archive.

- The **-v** option displays each file name as it processes the archive.
- The **-f** option uses a specified file name for the archive file.

After you have copied the archive file into an CMS record file you can delete it from the BFS.

Restoring a Complete Directory from a CMS Record File

To restore the directory in the previous example, enter one CMS command and one shell command:

```
OPENVM PUTBFS TESTPGM TAR A /tmp/posix/testpgm.tar (BFSLINE NONE
tar -xvf testpgm.tar
```

The **OPENVM PUTBFS** command copies the file containing an archive file into the specified directory in the file system. The **tar** command restores the contents of the archive so that they can be accessed in the file system. For the **tar** command, the **-x** option restores files from the archive.

Viewing the Contents of an Archive

To display the names of the files in the archive file, enter the shell command:

```
tar -tvf testpgm.tar
```

For the **tar** command:
- The **-t** option lists the contents of the archive on standard output.
- The **-v** option gives more detailed, or “verbose”, information on the list.

Restoring Selected Files from an Archive

To restore only selected files from the archive file, enter the shell command:

```
tar -xvf testpgm.tar file1.c file2.c tmpdir
```

This command restores only `file1.c`, `file2.c`, and all files in the directory `tmpdir` from the archive `testpgm.tar`.

With **tar**, it is not possible to restore only the files that are newer than the existing files. An alternative is to use the shell **pax** command with the **-u** option:

```
pax -ruf testpgm.tar
```

The **pax** command automatically recognizes that the archive is in TAR format.

Restoring Files Interactively

To restore files interactively, use the **-w** option on the shell **tar** command:

```
tar -xvwf testpgm.tar
```

With the **-w** option, the command displays each file name and waits for your response. Enter `y` to restore the file. If you enter any other character, **tar** skips over the file and continues processing.

Appending to an Archive

To back up a directory and append it to the end of an already existing archive, enter the shell command:

```
tar -rvf testpgm.tar /tmp/posix/testpgm2
```

tar appends all the files to an existing archive, even if some or all of the files already exist in the archive. You cannot do this if the archive file is a compressed archive file.

Now, to display the contents of the `testpgm.tar` archive, enter:

```
tar -tvf testpgm.tar
```

Backing Up Files Created over a Certain Number of Days

Suppose you want to back up all files that have been changed during last week. Use the shell command:

```
find /tmp/posix/testpgm -type f -mtime -7 | tar -cvf testpgm.tar -
```

- **-type f** tells **find** to select only files. This avoids duplicate input to **tar**.
- The **-** at the end of the **tar** command makes it read from standard input, which is the output of the **find** command.

Using pax to Back Up and Restore Files

pax can read and write files in CPIO ASCII format, CPIO binary format, TAR format, or USTAR format. It can read files that were written using **tar**, **cpio**, or **pax** itself. How it handles file name length and preservation of link information across the backup and restore process depends on the format you select: If you select CPIO, it behaves like the **cpio** command; if you select TAR, it behaves like the **tar** command.

For information on the CPIO and TAR archive file formats, see Appendix D.

In these examples, `/tmp/posix` is the working directory.

Backing Up a Complete Directory into a CMS Record File

To back up a complete directory, including the subdirectories and their contents, into a file, enter one shell command and one CMS command:

```
pax -wf archive_file directory_name
OPENVM GETBFS archive_file file_id (BFSLINE 1
```

where *directory_name* is the name of the directory you want to archive, *archive_file* is an absolute path name and *file_id* is a CMS record file specification. The **pax** command creates an archive file with the specified name in the current working directory. The **OPENVM GETBFS** command copies the archive file into the specified CMS record file.

For example, these two commands back up directory `/tmp/posix/testpgm` into file TESTPGM PAX:

```
pax -wf testpgm.pax /tmp/posix/testpgm
```

(next enter the CMS command)

```
OPENVM GETBFS /tmp/posix/testpgm.pax TESTPGM PAX (BFSLINE 1
```

For the **pax** command:
- The **-w** option writes to the archive file.
- The **-f** option lets you specify the name of the archive file.

After you have copied the archive file into an CMS record file you can delete it from the BFS.

Note: To avoid accidentally including the archive file you must do one of these:

- Explicitly specify the directories you want included (as in the above example)
- Write the archive file to a different directory than the one you are archiving. This example archives the contents of the current directory and writes the archive in another directory, `/tmp`:

  ```
  pax -w . > /tmp/pax.file
  ```

Restoring a Complete Directory from a CMS Record File

To restore the directory in the previous example, enter one shell command and then one CMS command:

```
OPENVM PUTBFS TESTPGM PAX A /tmp/posix/testpgm.pax (BFSLINE NONE
pax -rf testpgm.pax
```

The **OPENVM PUTBFS** command copies the file containing an archive file into the specified directory in the file system. The **pax** command restores the contents of the archive so that they can be accessed in the file system.

For the **pax** command, the **-r** option reads from the file specified with the **-f** option.

Working with a Compressed Archive

To compress the archive file when it is created, use **-z** option with the shell **pax** command:

```
pax -wzf testpgm.pax.z /tmp/posix/testpgm
```

Adding the `.z` to the file suffix `.pax` is a UNIX convention that indicates the file is compressed.

To restore the directory from the compressed archive:

```
pax -rzf testpgm.pax.z
```

Viewing the Contents of an Archive

To view the contents of archive, use the **-f** option with the shell **pax** command:

```
pax -zf testpgm.pax.z
```

If the file is not compressed, you do not need the **-z** option.

Specifying a Format for Backup

Use the **-x** option to specify format with the shell **pax** command. To create an archive file with a header in CPIO binary format:

```
pax -wzf testpgm.pax.z -x cpiob /tmp/posix/testpgm
```

To create an archive in USTAR format:

```
pax -wzf testpgm.pax.z -x ustar /tmp/posix/testpgm
```

Restoring Selected Files from an Archive

To restore a file from an archive file that is *not compressed,* you do not have to specify the file's format. However, when restoring a file from a compressed archive file, use the **-x** option to indicate the format of the archive file.

To restore only selected files from a compressed archive file, use this shell command:

```
pax -ruzf testpgm.pax.z -x cpiob file1.c file2.c tmpdir
```

This command restores only `file1.c`, `file2.c`, and all files in the directory `tmpdir` from the archive `testpgm.pax.z`. Because the **-u** option is specified, these files are restored only if they do not exist in the working directory or if the files in the working directory are older than those in the archive.

Restoring All But Selected Files from Backup

To restore all files except `file1` and `file2` use the following shell command:

```
pax -wczf testpgm.pax.z -x cpiob file1 file2
```

The **-c** option selects those files that do not match the pattern given on the command line. In this example, the pattern is the two file names.

Converting Between Code Pages

If you need to convert an archive file from one character set to another, use the **-o** option with the shell **pax** command:

```
pax -wf testpgm.pax -o from=IBM-1047,to=ISO8859-1 /tmp/posix/testpgm
```

This command backs up the `/tmp/posix/testpgm` directory, which is in the character set IBM-1047, into an archive file that is targeted to an ASCII character set (ISO8859-1).

This option is very useful for transferring text data between systems that use different code pages.

To restore from the archive file created in the previous example, enter the shell command:

```
pax -rf testpgm.pax
```

You do not have to specify **-o** option for restoring when the conversion was done at the time of backup.

A code page is also known as a code set. See the *z/OS: XL C/C++ Programming Guide* for more information about the code sets supported for this command.

Restoring an ASCII Archive File That Has Component Archive Files

Most archive files you receive are in ASCII format if they do not come from an OpenExtensions system. Therefore, you will need to convert the data from ASCII to EBCDIC when you restore the archive file.

This procedure becomes more complex if the archive file has component files that are archive files themselves. For example, let's say you have received an archive file for product A. The file is called `multfil.pax` and it has one embedded archive file, `pgma.pax`, along with "usual" component directories and files.

Assume that the main archive file is in the file system and is named `/usr/proda/multfil.pax`. Here is the sequence of steps:

1. Change to the directory `/usr/proda` and restore the main archive file, using the convert option; for example:

   ```
   cd /usr/proda
   pax -rf multfil.pax -o from=ISO8859-1,to=IBM-1047
   ```

where multfil.pax is the archive file name. No pattern is specified, because you want to restore all the members.

The component files are restored into a directory tree below your working directory (/usr/proda). The embedded archive file pgma.pax is also converted; as a result, it is unusable. You now need to restore it separately from the file multfil.pax.

2. Restore the component archive file from the original archive file without using the convert option. For example, you would restore pgma.pax in your working directory /usr/proda with this command:

   ```
   pax -rf multfil.pax pgma.pax
   ```

   This extracts pgma.pax from the archive file multfil.pax and overlays the garbled version from step 2. However, pgma.pax is still an archive file.

   Note: If pgma.pax is an archived directory that is a subdirectory of the directory that multfil.pax has archived, include that subdirectory name in the path name when you restore it; for example:

   ```
   pax -rf multfil.pax pgm/pgma.pax
   ```

3. Restore the component archive file, /usr/proda/pgma.pax, using the convert option. In our example, you can now convert the archive file pgma.pax:

   ```
   cd /etc/proda
   pax -rf /usr/proda/pgma.pax -o from=ISO8859-1,to=IBM-1047
   ```

   This becomes a separate directory tree below your working directory /etc/proda.

   If there are multiple component archive files in an archive file, you may want to write a shell script to restore them.

Chapter 13. 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 a user creates and processes a user runs. When you create a directory, the directory is automatically associated with your UID, and its GID is set to the owning GID for the *parent directory* (the directory it is in). When you create a file, the file is automatically associated with your UID, and its GID is set to the owning GID for the parent directory.

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 any 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*.) There are three types of permissions that you can grant to each class of user. 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 or process	**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.)

Default Permissions Set by the System

You can override the defaults by setting the system mask using **OPENVM SET MASK**.

The following table shows the default permissions set by the system:

Task	Using	Default Permissions
Create a directory	**mkdir** shell command	owner=rwx group=r-x other=r-x

Task	Using	Default Permissions
Create a directory	**OPENVM CREATE DIRECTORY**	owner=rwx group=r-x other=r-x
Create a file	**XEDIT command**	owner=rw- group=r-- other=r--
Create a file	**ed** editor	owner=rw- group=r-- other=r--
Create a file	Redirection (**>**)	owner=rw- group=r-- other=r--
Create a file	**cp** command	Sets the output file permissions to the input file permissions.
Create a file	**OPENVM GETBFS command**	If any execute permissions are on in the source file: owner=rwx group=r-x other=r-x If no execute permissions are on in the source file: owner=rw- group=r-- other=r--
Create a file	**OPENVM PUTBFS**	If the source file is a BFS file and any of its execute permissions are on, or if the source file is a CMS record file in MODULE format: • owner=rwx • group=r-x • other=r-x Otherwise, default permissions are: owner=rw- group=r-- other=r--

Changing Permissions for Files and Directories

Using CMS

You can use the **OPENVM PERMIT** command to change permissions for your files and directories. To change permissions, you must be the owner or a superuser.

For example, to add public read and write permission to the file `fileaa`, enter:

```
OPENVM PERMIT fileaa --- --- rw- (ADD
```

To remove public write permission, enter:

```
OPENVM PERMIT fileaa --- --- -w- (REMOVE
```

To replace the permission bits with an entirely new set of permission bits, use the **(REPLACE** option of **OPENVM PERMIT**, for example:

```
OPENVM PERMIT fileaa rw- rw- rw (REPLACE
```

Note that **(REPLACE** is the default.

See *z/VM: OpenExtensions Commands Reference* for more information.

Using the Shell

You can use the **chmod** command to set or change permissions for your files and directories. To change permissions, you must be the owner. If you are uncertain about ownership, use the **ls -l** command and see if your CMS user ID is in the third field.

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 *z/VM: OpenExtensions Commands Reference*.

Using a Symbolic Mode to Specify Permissions

A symbolic mode has the form:

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.

The *permission* part of a symbolic mode is any combination of the following:

r Read permission. If this is off, you cannot read the file.

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 132 for more information.

t This sets the *sticky bit* on. The sticky bit is supported for compatibility only. OpenExtensions takes no special action in support of this bit.

w Write permission. If this is off, you cannot write to the file.

x Execute permission. If this is off, you cannot process the file.

X Execute/search permission if the specified object is a directory or if the current mode bits have at least one execute/search bit set. If the object is not a directory and if none of the execute/search bits are set in the current mode then **X** is ignored.

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
```

You can specify multiple symbolic modes if you separate them with commas.

Using Octal Numbers to Specify Permissions with the Shell

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 process 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 specifying the first octal digit, you can specify 3 digits instead of 4. When the first digit is not specified, some typical 3-digit permissions are specified in octal this way:

Table 2. Three-Digit Permissions Specified in Octal

Octal Number		Meaning
666		owner (rw-) group (rw-) other (rw-)
700		owner (rwx) group (---) other (---)
755		owner (rwx) group (r-x) other (r-x)
777		owner (rwx) group (rwx) other (rwx)

Displaying File and Directory Permissions

Using CMS

Use the **OPENVM LISTFILE** command with the **(OWNERS** option to display permissions for files and directories.

```
OPENVM LIST / (OWNERS SUBD
Directory = '/'
User ID    Group Name  Permissions Type  Path name component
bfs1       CMSUSRS     rwx rwx rwx  D    'International_data'
bfs1       CMSUSRS     rwx rwx rwx  D    'US_cities_and_towns'
bfs1       CMSUSRS     rw- rw- rw-  F    'US_cities_and_towns/Anaheim_File'
bfs1       CMSUSRS     rw- rw- rw-  F    'US_cities_and_towns/Boston_File'
bfs1       CMSUSRS     rw- rw- rw-  F    'US_cities_and_towns/Detroit_File'
bfs1       CMSUSRS     rw- rw- rw-  F    'US_cities_and_towns/Endicott_File'
Ready;
```

Using the Shell

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  nettle  groupz   96 Jun 15 10:34 statrp
-rwx------  fff---    1  nettle  groupz  107 Jul 10 07:45 feb95
-rwx------  fff---    1  nettle  groupz   80 Aug 10 13:15 aprl95
-rwx------  fff---    1  nettle  groupz  150 Jan 13 10:45 jun94
drwxr-xr-x  fff---    2  nettle  groupz   96 Jun 17 09:05 dbappl
-rwxr-x---  fff---    1  nettle  groupz  150 Jun 17 10:15 txtn1
```

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)
s for a socket 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` in the final field (`--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 through the BPX1CHA callable service; 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 (CMS user ID).

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 file name **lnk** in this example:

```
lrwxrwxrwx   1 nettle    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*. 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 attempt to set those permissions. When a file or directory is created, the permissions set by the program are adjusted by the *file mode creation mask* value: the final permissions set are the program's permissions minus what the file mode creation mask restricts.

Using CMS

Set the file mode creation mask using **OPENVM SET MASK**. This sets the mask for the current process and for any new processes created until your CMS virtual

machine is IPLed or another **OPENVM SET MASK** command is issued. You can put this command into your `PROFILE EXEC` file so that it is always in effect.

For example, to deny write access to users other than the file owner you may specify:

```
OPENVM SET MASK rwr R-X R-X
```

A subsequently created file would have these file permissions initially:

```
Owner    Group    Public
 rwx      r-x      r-x
```

Using the Shell

You can set the 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.

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 enter just **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 OpenExtensions shell's initial setting of the mask is 022, which means that read, write, and execute permission is set on for the owner, and read and execute permission is set on for group and other.

To make your **umask** setting take effect each time you start the shell, put the **umask** command in `$HOME/.profile`.

Changing the Owner ID or Group ID Associated with a File

You 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:

- The superuser can change the owner (UID) of a file
- The superuser or the file owner can change the group (GID) for a file. The file owner must have the new group as his group or one of his supplementary groups.

Using CMS

Use the **OPENVM OWNER** command to change both the owner (UID) and the group (GID) for a file. To change just the group for `filea`, enter:

```
OPENVM OWNER filea newgroup
```

To change the owner and the group, enter:

```
OPENVM OWNER filea newgroup newowner
```

To change just the owner, enter:

```
OPENVM OWNER filea * newowner
```

In these examples, *newowner* and *newgroup* are the VM user ID of the user who's to be the new owner of the file and the POSIX group name with which the file is to be associated.

Using the Shell

To change the owner (UID) of a file, enter a **chown** shell command. To change the group (GID) of a file, enter a **chgrp** command.

Superuser tasks are discussed in *z/VM: OpenExtensions Commands Reference*.

Temporarily Changing the User ID or Group ID during Execution

An executable file, which is a file containing a shell script or a program, can have an additional attribute. This permission setting allows a program temporary access to files that are not normally accessible to other users.

Using CMS

When you use the **OPENVM LISTFILE** command with the **(OWNER** option, you see three columns indicating the permissions for owner, group, and public as discussed earlier. For the owner or group, the execute column (x) may have an 's', indicating that the UID or GID will be set on execution of this file. An 's' implies execute authority.

Using the Shell

When you enter **ls -l**, 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.

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.

To give a file the set-user-ID permission, use the **chmod** command. For example,

```
chmod u+s cmd1
```

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.

Chapter 14. Editing Files

You have a choice of three editors for creating and changing files:

- XEDIT, a full-screen editor that you call from CMS or the shell
- The **ed** editor, a line editor that you can call from the shell or CMS
- The **sed** stream editor, a noninteractive editor. It is intended for *systematic* editing; you call the editor with a file of editing commands and a target data file and it produces an edited target file, with no user interaction.

Using XEDIT to Edit a BFS File

Using XEDIT

XEDIT provides a full-screen editor you can use to create and edit BFS files.

Using XEDIT, you can edit only regular files (not special files). You need read permission for the file and search permission for any intermediate directories. You need write permission to save changes to the file.

When you create a new file, you must have the appropriate permissions to add a new file to the parent directory. When XEDIT creates a file, it attempts to set the permission bits to `rw- r-- r--`; if you have a umask in effect, those bits will be downgraded accordingly. See “Setting the File Mode Creation Mask” on page 130 for more information.

XEDIT allows many editing sessions at a time. 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 XEDIT 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 `"` on the line number. You can enter line commands for several lines at the same time.

XEDIT subcommands To perform editing tasks, you enter *XEDIT subcommands*. For example, you can use the **LOCATE** subcommand to scan data for a specific character string. If you entered:

```
LOCATE /printf(
```

on the command line, the editor locates the next occurrence of **printf(**. Likewise, you can enter the **CHANGE** subcommand to make global changes within a file. For example:

```
CHANGE /CRTL/C-RTL/ *
```

changes all instances of `CRTL` to `C-RTL`.

You can use other XEDIT subcommands, such as **GET**, **PUT**, **PUTD**, **FILE**, and **SAVE**, to work with other files from within your XEDIT session.

CMS commands While you are editing one file, you can use CMS commands to work with other files, or to perform other tasks.

To end an edit session:
- Saving all changes, enter the **FILE** subcommand.
- Without saving any changes, enter the **QQUIT** subcommand.

When you end the edit session, you go back to where you were when you began it: on the entry panel, to the CMS `Ready;` prompt, or to the shell prompt.

All You Ever Wanted to Know about XEDIT

The discussion in this chapter is an introduction to XEDIT. For detailed information about XEDIT, including the subcommands just mentioned, use the online HELP facility or see *z/VM: XEDIT Commands and Macros Reference*.

Support for Doublebyte Characters

The XEDIT command works with doublebyte characters. An XEDIT subcommand, **SET ETMODE** (for extended mode) controls whether XEDIT recognizes DBCS strings. The initial setting is based on whether the terminal can display double-byte characters.

Code Page Conversion

When you edit a BFS file using XEDIT, 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 will be stored as characters that will not be properly interpreted by the C/C++ compiler, shell, or utilities. For a discussion of code page conversion, see “Understanding Code Page Conversion” on page 20.

Typing Tabs using XEDIT

Writing makefiles for the **make** utility requires the use of a <tab> character. **awk** programs can also use tabs.

If you are using an OpenExtensions editor (such as **ed**), you can type a tab character as an <EscChar-I> sequence. When you press **ENTER**, a blank space is displayed.

If you are using XEDIT, you cannot type a tab character (XEDIT handles only displayable characters). Instead, you can:

1. Select a substitute for a tab character, for example, the character @.
2. Whenever you want a tab to appear, type an @ instead of a tab character.
3. When you have finished editing the file, on the command line enter:

```
top
alter @ 05 * *
```

   This converts all @ in each line to the hex character 05, which is a tab.

The foregoing is just one of several methods you can use to edit hex data.

If you use XEDIT to edit an existing file that has tabs in it:

1. If you make no changes to a record, the tabs in that record are preserved.
2. If you make a change on a record that contains TAB characters, then XEDIT expands the tabs to spaces according to the settings established by XEDIT's **SET TABS** subcommand.

Preserving Trailing Blanks in Files

XEDIT removes trailing blanks from lines in files unless the **BFSLINE** option or **SET BFSLINE** subcommand is used. If this is undesirable, use the **sed** stream editor instead.

Working with Lowercase or Mixed-Case Files

You control whether to convert lowercase characters to uppercase when you type in a file. The initial setting is based on the file type of the file being edited. Because BFS files do not have a file type the initial setting will be in mixed-case. To change characters entered in lowercase to uppercase, enter the XEDIT subcommand **SET CASE UPPERCASE**, or simply **CASE U**. After entering the command, all characters will be uppercase. Changing the setting after entering data, however, will not affect characters already entered.

Accessing a File to Edit

To use XEDIT to edit a regular file enter `XEDIT` *pathname* (`NAMETYPE BFS`. The path name specified may be a relative, absolute, or fully-qualified path name. It may also be mixed-case. If the path name contains characters such as blanks, parentheses, or X'FF', it must be enclosed in quotation marks.

Since XEDIT is used to edit both CMS record file (which use a naming convention of *filename filetype filemode*) and BFS files (which use *pathnames*), you must tell XEDIT which naming convention you are using. Do this using the **NAMETYPE CMS** or **NAMETYPE BFS** option on the XEDIT command.

Another XEDIT option, **BFSLINE**, lets you tell the editor how to translate the byte stream into lines or *records*. You can select to separate it into records of a fixed length using **BFSLINE** *lrecl* or you can separate them into records using one or two line end characters. You can select X'15' for a line end character by using the **BFSLINE NL** option, or you can specify the line end character(s) using a character string or hexadecimal string. The **BFSLINE NL** option is the default if **BFSLINE** is not specified.

You can also use a *profile* to set the **NAMETYPE**, **BFSLINE**, or other XEDIT options. An XEDIT profile is a CMS record file. See *z/VM: XEDIT Commands and Macros Reference* for more information.

Working with Other Files While Editing a File

While editing a BFS file, you can type XEDIT subcommands or other CMS commands on the command line. Descriptions of some of those commands follow:

Table 3. Sample XEDIT Subcommands

GET	Copies another BFS file, CMS record file, or part of a CMS record file into the file you are editing.
PUT	Appends all or part of the file you are editing to a BFS or CMS record file. (If the file being written to does not exist, it is created.)
PUTD (PUT Delete)	Appends all or part of the file you are editing to a BFS or CMS record file, and removes the lines from the file you are editing. (If the file being written to does not exist, it is created.)
FILE	Writes the file you are editing out into the same or a different BFS or CMS record file, and ends the editing session.
SAVE	Writes the file you are editing out into the same or a different BFS or CMS record file, and continues the editing session.
XEDIT	Edits another BFS or CMS record file during your current edit session.

Using Edit Macros

If you use XEDIT macros that depend on file attributes, you may need to tailor the macros to work with the assumptions made about a BFS file. If you use the **BFSLINE** *lrecl* option, the file looks like a *fixed record format* CMS record file while in the XEDIT session. If you use any other **BFSLINE** option, the file looks like a *variable record format* file while in the XEDIT session. For more information, see *z/VM: XEDIT Commands and Macros Reference*.

Copying into a File

Use the **GET** subcommand to copy an entire BFS file into the file you are editing. You can also use it to copy all or part of a CMS record file into the file you are editing.

1. Use a slash (/) in the prefix area of the line you want the new data to follow. This makes that line the *current line*.
2. Specify `GET` *pathname* or `GET` *fn ft fm* on the command line. (Note that you may need to issue **SET NAMETYPE BFS** or **SET NAMETYPE CMS** first.)
3. Press **ENTER**.

Moving Data into a File

Use the **PUT** or **PUTD** subcommand to put all or part of the file you are editing into a BFS file or a CMS record file. If you use **PUTD**, the lines are removed from the file you are editing.

1. Use a slash (/) in the prefix area of the first line you want written. This makes that line the *current line*.
2. Specify `PUT` *nn pathname* or `PUT` *nn fn ft fm* on the command line. *nn* tells XEDIT how many lines you want to write. (Note that you may need to issue **SET NAMETYPE BFS** or **SET NAMETYPE CMS** first.)
3. Press **ENTER**.

Replacing a File or Creating a File with Data from a File

Use the **FILE** or **SAVE** subcommand to replace the contents of a file with the file you are editing.

1. On the command line type one of the following:
   - Type `FILE` to save the file under the current name.

- Type `FILE` *pathname* to file the name to a BFS file. (Note that **NAMETYPE** must be set to **BFS**.)
- Type `FILE` *fn ft fm*. (Note that **NAMETYPE** must be set to **CMS**.)

2. Press **ENTER**.

You may perform the same function with the **SAVE** subcommand. However, in that case, you do not exit from your XEDIT session.

Editing Another File During an Edit Session

Use the **XEDIT** subcommand during your current editing session to edit another BFS or CMS record file. If the file does not exist, it is created.

1. Type `XEDIT` *pathname* (`NAMETYPE BFS` or `XEDIT` *fn ft fm* on the command line and press **ENTER**. (Note that unlike other subcommands that use path names, the **XEDIT** subcommand does not use the **NAMETYPE** setting in effect for the current session. In order to change from the default **NAMETYPE CMS**, you must specify **NAMETYPE BFS** on the **XEDIT** subcommand, or in your `XEDIT PROFILE`.)
2. Press **ENTER**.

Edit Recovery

You can determine the level of recovery for your XEDIT session using the **SET AUTOSAVE** command. When the automatic save function is in effect, the editor automatically issues a **SAVE** subcommand each time the specified number of alterations is made. The AUTOSAVE file is written to a CMS minidisk or accessed SFS directory. See the **SET AUTOSAVE** command in *z/VM: XEDIT Commands and Macros Reference* for more information about how to set the level of recovery and how to perform the recovery.

Using the ed Editor

Using the Shell

ed is a line editing program available in the OpenExtensions shell. 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 *z/VM: OpenExtensions Commands 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 file name 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. 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 do not 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:

*n*r *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,
Take time to work, work is the price of success.
Take time to think, thoughts are the source of power.
```

After you know the line numbers, you could insert the file `addlines` after the line `Take time to think...`. Thus, you would enter:

```
2r addlines
```

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:

  ```
  ns/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,a2s/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,a2s/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:

  ```
  ns/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:

  ```
  ns/^/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:

*a1*t*n*

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,*a2*t*n*

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,*a2*m*n*

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:

!*commandname*

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 a BFS 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 *z/VM: OpenExtensions Commands Reference*.

Chapter 15. 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 VM system intentionally screens the user from printer knowledge and uses a pool of printer resources.

You can, of course, download BFS files and print them at your workstation. However, it may be more convenient to have the shell send print jobs to VM system printers. In addition, you may want to use the large-volume printing facilities offered by VM.

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 BFS 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 *z/VM: OpenExtensions Commands 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 40 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 printer:

```
lp filename
```

You can specify more than one file name with the command. The **lp** command uses existing VM 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.

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.

Printing with CMS Commands

Using CMS

Many VM systems have elaborate printing facilities based on Advanced Function Printing™ products. These facilities are generally accessed through REXX execs provided by the system administrator. They require the files to be printed to be either on accessed minidisks or SFS directories. To use these printing facilities, first copy the file from BFS to a minidisk or SFS directory. For example if you want to print the file `output.list` that resides in your working directory, issue the command:

```
openvm getbfs output.list output list
```

Then, invoke your installation's printing utility to print the file.

For simple printing of character files, CMS provides the PRINT command. Once you have the file on a minidisk on SFS directory, you can print the file `OUTPUT LIST` through the following command:

```
print output list
```

The attributes and status of your print file can then be checked via the command:

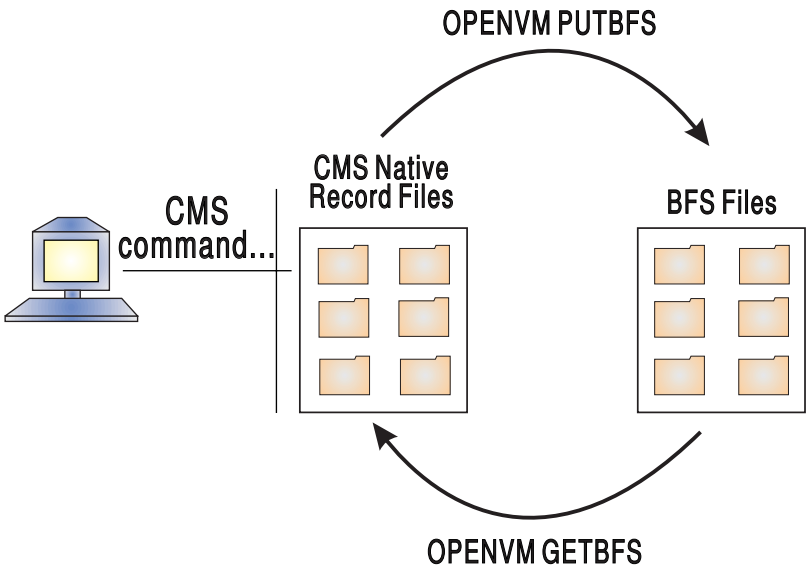
```
query print
```

Chapter 16. Copying Files

You can copy:

- CMS record files into the byte file system (BFS).
- BFS files into CMS record files.
- BFS files into other BFS files.

BFS files can be files in NFS-mounted remote file systems.



To move data between BFS files and CMS native record files, use these commands:

OPENVM PUTBFS
: Puts (copies) a CMS record file into the BFS. You can select code page conversion for singlebyte data.

OPENVM GETBFS
: Gets a BFS file and copies it into a CMS record file. You can select code page conversion for singlebyte data.

To read about **OPENVM GETBFS** and **OPENVM PUTBFS** commands, see *z/VM: OpenExtensions Commands Reference*.

Executable Modules. You can also copy executable modules into and out of the BFS using the **OPENVM GETBFS** and **OPENVM PUTBFS** commands.

Copying a CMS Record File into a BFS File

You might want to copy a CMS record file to a BFS file so that:

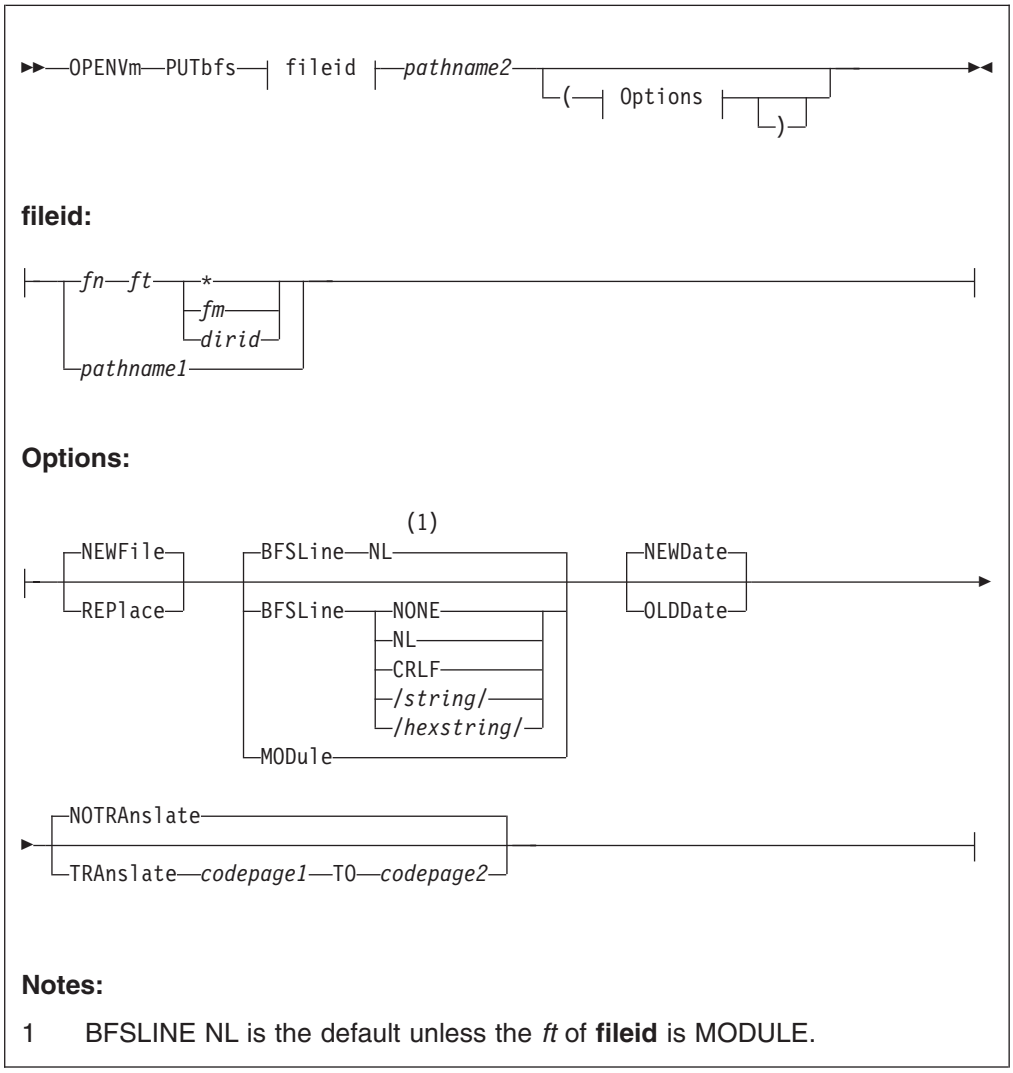
- The data can be used by a program running under the shell.
- You can compile and build it in the shell using the **make** command.

If you are moving the file permanently to the BFS, use the CMS **ERASE** command to delete the file from the record file system after copying it.

Use the CMS **OPENVM PUTBFS** command to do the copy. You can enter **OPENVM PUTBFS** in CMS, in the shell, or in XEDIT. See “Where You Can Enter a CMS Command” on page 94 for information on entering CMS commands in CMS, the shell, and XEDIT.

OPENVM PUTBFS

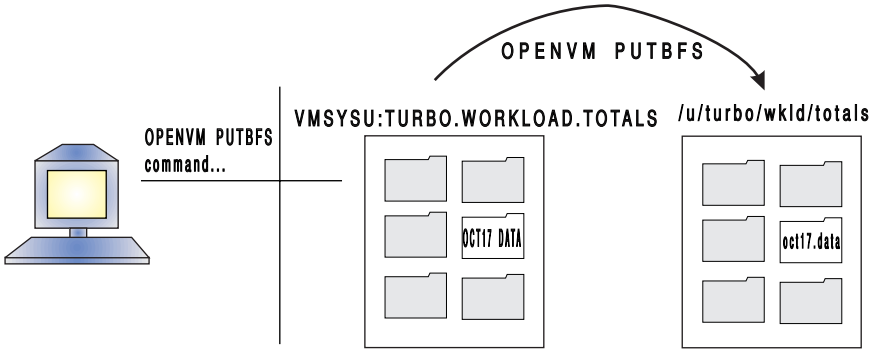
The **OPENVM PUTBFS** command syntax is:



Notes:

1 BFSLINE NL is the default unless the *ft* of **fileid** is MODULE.

Example: Using OPENVM PUTBFS with a CMS Record File



If the user that has the user ID of TURBO wants to copy a CMS record file into a BFS file, he might enter the following CMS **OPENVM PUTBFS** command:

```
openvm putbfs oct17 data vmsysu:turbo.workload.totals /u/turbo/wkld/totals/oct17.data (TRANSLATE nnn TO 1047
```

This command:

- Copies the CMS record file OCT17 DATA from VMSYSU:TURBO.WORKLOAD.TOTALS to a text file with the path name `/u/turbo/wkld/totals/oct17.data`.
- Converts the data from the VM country-extended code page *nnn* to code page IBM-1047. If you do not want conversion, omit the TRANSLATE operand; For more information, see “Understanding Code Page Conversion” on page 20.
- Sets a default mode based on the current setting of the file mode creation mask (read-write-execute permission) if `oct17.data` is a `new` file.

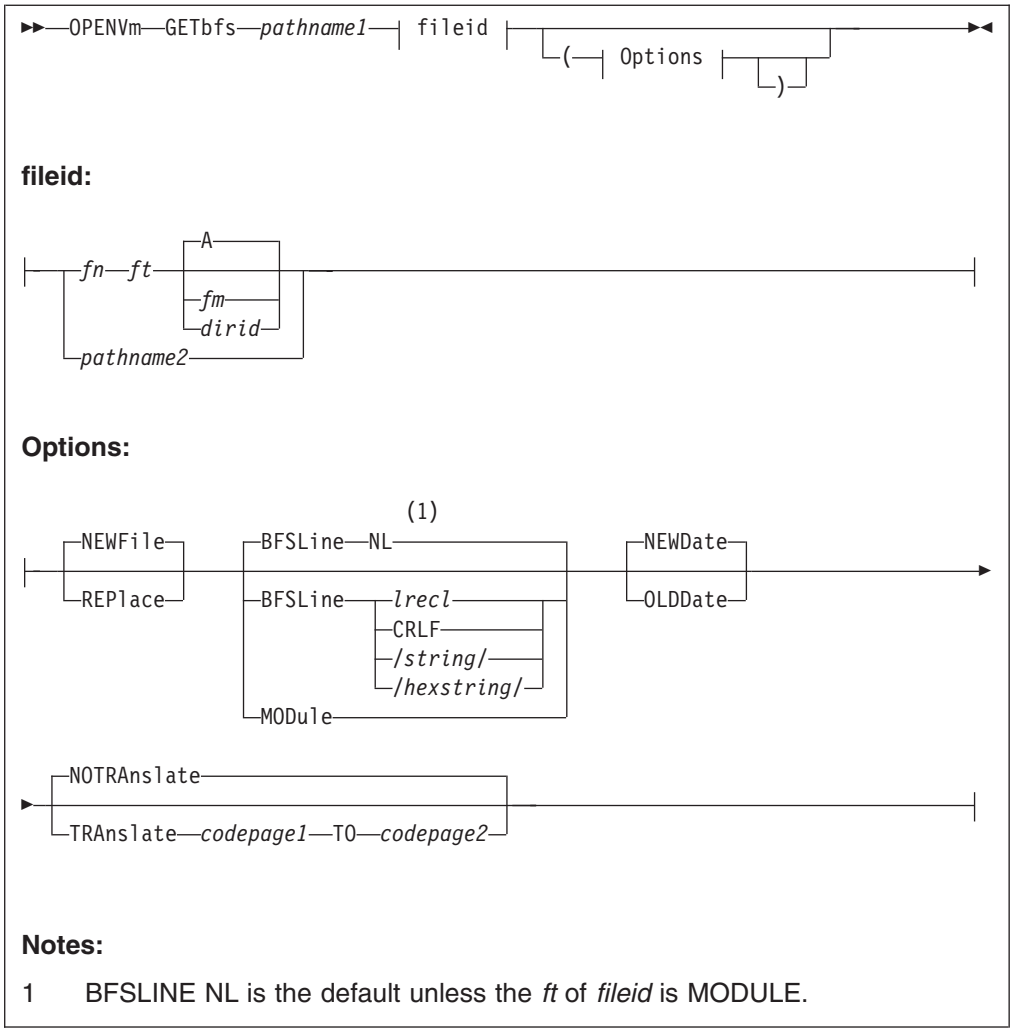
Copying a BFS File to a CMS Record File

You might want to copy a BFS file into a CMS record file

The OPENVM GETBFS command copies a BFS file into another BFS file, an SFS directory, or onto a CMS minidisk.

OPENVM GETBFS

The **OPENVM GETBFS** command syntax is:



For a complete command descriptions see *z/VM: OpenExtensions Commands Reference*.

Copying a BFS File to Another BFS File

You can use the shell command **cp** or the CMS commands **OPENVM GETBFS** or **OPENVM PUTBFS** to copy files within the BFS.

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 `a` and `b` into the directory called `dir3`. The copied files have same the file names as the original, so you will find files `a` and `b` in the directory `dir3`.

For further information on the **cp** command, see *z/VM: OpenExtensions Commands Reference*.

Chapter 17. Transferring Files between Systems

You may typically create applications and files at your workstation and then move the resulting files to the byte file system (BFS) for further application development—such as compiling and debugging or to share the files. There may also be times when you want to send BFS files to your workstation.

To move a file or file system between your workstation and the BFS, you can use one of these:

- The File Transfer Protocol (FTP) facility of TCP/IP, when both the workstation and VM system have TCP/IP installed.
- The SEND and RECEIVE programs available 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 CMS. If you are working in the shell, use **exit** to return to CMS command mode *before* using the programs.

Transferring to the Byte File System

You can use NFS to mount the file system in your byte file system. Then you can use the files as if they were in the local BFS. Or you can perform the following procedure.

First transfer the file to the host. Then, while working at the host, perform these steps:

1. Copy the file from the CMS record file system into the BFS, using the **OPENVM PUTBFS** command.

   If you need to convert to a shell-supported code page, use the TRANSLATE option on the **OPENVM PUTBFS** command. See “OPENVM PUTBFS” on page 148.
2. If desired, after the copy you can delete the CMS record file with the CMS **ERASE** command.

Transferring a File to the Workstation

If you have an NFS client on your workstation, you can use NFS to mount the byte file system so that it appears to be a drive on your workstation.

If you do not have an NFS client on your workstation, then at the host, perform these steps:

1. Copy the BFS file to an CMS native record file using the **OPENVM GETBFS** command.

   Singlebyte data: If you need to convert to a different code page, you can use the **(TRANSLATE** option on the **OPENVM GETBFS** command.
2. If desired, after the copy you can delete the BFS file with the **rm** shell command or the **OPENVM ERASE** command.
3. Then transfer the file to the workstation.

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 116. This section discusses the steps involved in:

- Installing an archive file from tape or diskette into an BFS file system
- Putting an archive file on tape or diskette to send to another site

Putting an Archive File into a Byte 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 BFS file system:

1. Transferring the archive file into a CMS record file from a workstation.
2. Copying the archive file into the byte file system (BFS).

Step 1. Transferring the Archive File to a Record File

From a Workstation: If you have TCP/IP on your workstation, you can use the File Transfer Protocol (FTP) command to transfer an archive file to VM. *At the workstation*:

a. Copy the archive file into a file from one of these:
   - A diskette for a PS/2 or an RS/6000®
   - A tape for an RS/6000

b Use a file transfer tool (such as **FTP**) to copy the file from the workstation to VM in binary mode.

c. Go to “Step 2. Copying the File into the Byte File System.”

From a Tape Drive on Your VM System: If you have an archive file on tape and the necessary tape drive at your VM system, you can copy the file directly from the tape into a BFS directory.

Working at VM:

a. Copy the archive file from the tape into a BFS directory using the **OPENVM PARCHIVE** command.

b. Go to Step 2. Copying the File into the Byte File System, which follows.

Step 2. Copying the File into the Byte File System

a. If the archive file has been placed in the record file system, use the **OPENVM PUTBFS** command with the **(BFSLINE NONE** option to copy the file into the BFS. See “OPENVM PUTBFS” on page 148 for more information. The archive file becomes a single file in the file system.

b. Use the **pax**, **tar**, or **cpio** 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 OpenExtensions shell, use the **pax** command with the **–o** option. See “Backing Up and Restoring Files: The Options” on page 116 for more information.

Sending an Archive File to Others

You may want to send an archive file on tape or diskette. There are two major options available.

1. A diskette or tape at a workstation
2. A tape at your VM system

For both options you must first use the **pax**, **tar**, or **cpio** shell command to create the archive file for a directory or file system. All the component files are stored in one archive file.

If you need to convert to a different code page than the one used in the OpenExtensions shell, use the **pax** command with the **–o** option. See “Backing Up and Restoring Files: The Options” on page 116 for more information.

Option 1. Copying to a Diskette or Tape at a Workstation

a. Create the archive file as described above.

b. Use **OPENVM GETBFS** to move the file to CMS file system.

c Use a file transfer tool (such as **FTP**) to copy the file from VM to the workstation in binary mode.

d. At the workstation, copy the archive file onto one of these:
   - A diskette, for a PS/2 or RS/6000
   - A tape, for an RS/6000

Option 2. Transferring the Archive File to a Tape at the Host

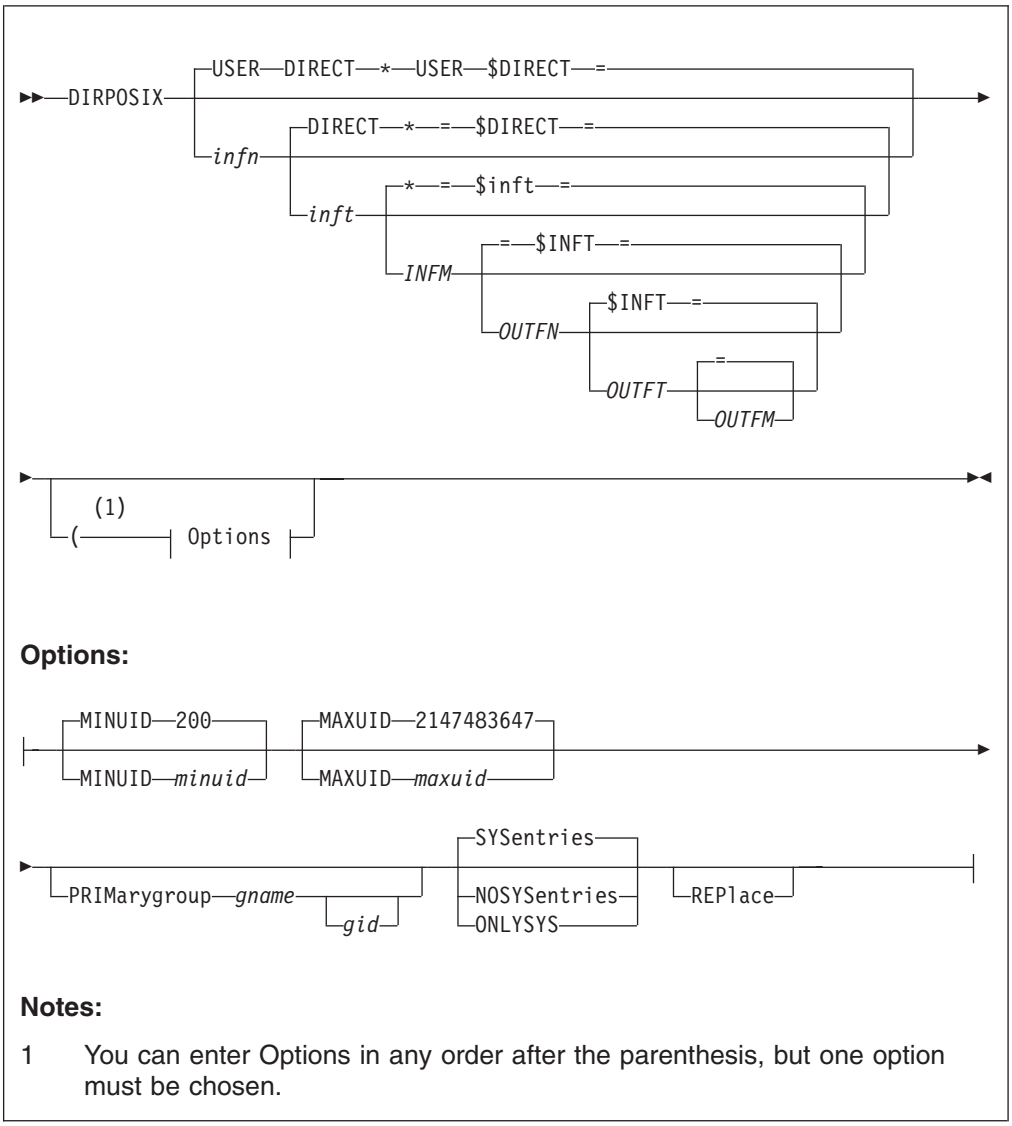
a. Create the archive file as described above.

b. Use the **OPENVM PARCHIVE** command to copy the archive file from the BFS to the tape.

Part 4. Appendixes

Appendix A. DIRPOSIX Utility

DIRPOSIX



Purpose

Use the DIRPOSIX utility to add POSIX information to a user directory source file. It performs the following functions:

- Assigns a unique UID to each userid that has no UID specification and is not listed in the DIRPOSIX USEREXCL file. See usage note 10 on page 162 for more information.
- Assigns a primary group to each userid that has no primary group specification and is not listed in the DIRPOSIX USEREXCL file.
- Adds the standard “system” group definitions and the standard “system” user definitions, if they do not already exist. See usage note 12 on page 162 for more information.

DIRPOSIX provides a mechanism for reserving installation-specified UIDs; it will not assign any UIDs listed in the DIRPOSIX UIDEXCL file. See usage note 11 on page 162 for more information.

Operands

infn
is the file name of the input user directory file. The default is USER.

inft
is the file type of the input user directory file. The default is DIRECT.

infm
is the file mode of the input user directory file. If it is not specified, the default file mode is asterisk (*), and DIRPOSIX uses the CMS file search order to locate the input user directory file.

outfn
is the file name of the output user directory file. If *outfn* is omitted or specified as '=', the output file has the same file name as the input file.

outft
is the file type of the output user directory file. If *outft* is specified as '=', the output file has the same file type as the input file. The default is described by *$inft*.

$inft
is the default file type of the output user directory file. The name is the result of the concatenation of '$' with the first 7 characters of the file type of the input file.

outfm
is the file mode of the output user directory file. The default is =. If *outfm* is omitted or specified as '=', the output file has the same file mode as the input file.

MINUID *minuid*
specifies the minimum UID that DIRPOSIX is permitted to assign. It is the lower bound of the UID range that DIRPOSIX uses when determining the first UID to be assigned. No UIDs below *minuid* will be assigned by DIRPOSIX. *minuid* must be between 10 and 4294967000 inclusive, and it must be no larger than *maxuid*. The default MINUID value is 200.

MAXUID *maxuid*
specifies the maximum UID that DIRPOSIX is permitted to assign. It is the upper bound of the UID range that DIRPOSIX uses when determining the first UID to be assigned. No UIDs above *maxuid* will be assigned by DIRPOSIX. *maxuid* must be between 10 and 4294967000 inclusive, and it must be no smaller than *minuid*. The default MAXUID value is 2147483647 (X'7FFFFFFF'). 4294967001 to 4294967293 are reserved for installation use. They will not be assigned by DIRPOSIX.

PRIMarygroup *gname gid*
specifies the primary group to be assigned to each userid that has no primary group specification and is not listed in the DIRPOSIX USEREXCL file. DIRECTXA and DIRPOSIX support mixed case POSIX group names, so care should be taken when specifying *gname*.

If *gid* is not specified, then the source directory file must already contain a POSIXGROUP definition for *gname*. If both *gname* and *gid* are specified and the source directory file does not already contain a POSIXGROUP definition for

gname, then DIRPOSIX adds a POSIXGROUP statement to the directory to define the group. The *gid* operand must specify a number between zero and 4294967295.

SYSentries
specifies that DIRPOSIX should add the standard system group definitions and the standard system users. See usage note 12 on page 162 for more information.

NOSYSentries
specifies that DIRPOSIX should not add the standard system group definitions nor the standard system users.

ONLYSYS
specifies that DIRPOSIX should add the standard system group definitions and the standard system users along with their associated values, but no other changes should be made. See usage note 14 on page 163 for more information.

REPlace
indicates that DIRPOSIX may overwrite an existing output file with the new output file. The input file cannot be the same as the output file.

A temporary file is created by the name of DIRPOSIX $TEMPDIR. As a result, this name cannot be used as an input or output file name.

Usage Notes

1. For a complete description of the directory control statements, see *z/VM: CP Planning and Administration*.
2. The DIRPOSIX utility can be used to convert a non-POSIX directory file to one that contains basic, meaningful POSIX information. It may be run multiple times against the same source directory; this is useful for adding POSIX information to the directory entries of newly added userids.
3. The source directory file to be processed may be in either the monolithic format or the cluster format. The directory file created by DIRPOSIX is always in the monolithic format. For cluster format, the name of the index file is specified as the input file.
4. Users with no POSIX information in their directory entry have UIDs and primary group specifications assigned by DIRECTXA. DIRECTXA assigns default values to these users. See *z/VM: CP Commands and Utilities Reference* for details on the default values.
5. DIRPOSIX ignores PROFILE definitions. They are not assigned UIDs or primary groups by DIRPOSIX, and they are not considered when determining if a user has a UID or primary group specification. DIRPOSIX adds a POSIXINFO statement to each user that needs data assigned. Less source directory DASD space would be utilized if the data that is common to multiple users was added within a profile.
6. DIRPOSIX ignores POOL users. They are not assigned UIDs or primary groups by DIRPOSIX.
7. By default, DIRPOSIX reserves certain ranges of UIDs for the directory administrator to use as desired. The MINUID and MAXUID specifications can be used to reserve different ranges. They can also be used to permit DIRPOSIX to use a larger UID range than the default values.

8. DIRPOSIX will assign UIDs beginning one past the largest UID currently in use in the range of available UIDs. The only exception to this rule is that the special POSIX users are assigned specific UIDs rather than the next available UID.
9. DIRPOSIX can be used to assign a specific UID to a certain userid. If all but one of the users in the directory already have UIDs or are listed in the DIRPOSIX USEREXCL file, then only that one userid will be processed by DIRPOSIX. By specifying the same value for MINUID and MAXUID, DIRPOSIX is forced to assign that UID to the user. This technique is useful for assigning a specific UID to a new userid.
10. The DIRPOSIX USEREXCL file can be used to prevent DIRPOSIX from adding POSIX information to certain users' directory entries. All users who should not have POSIX information must be listed in the DIRPOSIX USEREXCL file prior to running DIRPOSIX.

    DIRPOSIX USEREXCL allows one or more userids to be specified on each line. If more than one userid is listed, one or more blanks must separate each entry. If asterisk (*) is the first non-blank character, the whole line is treated as a comment. If there is a need to have a userid begin with an *, place a userid that does not have a beginning * first on the same line in the file.
11. The DIRPOSIX UIDEXCL file can be used to prevent DIRPOSIX from reusing UIDs that were once assigned but are now available. Because the ownership of POSIX files is based on UIDs, if a user who has created POSIX files is later deleted from the directory, and that user's UID is reused by another userid, then the other userid has become the owner of the files. Any other privileges based on UID will be similarly granted to the other userid. Reusing UIDs is not recommended without careful consideration of the potential implications.

    Installations that use DIRPOSIX to assign UIDs to new users should consider using the DIRPOSIX UIDEXCL file to prevent the reuse of UIDs of deleted userids. When a userid is deleted from the directory, that user's UID could be added to the DIRPOSIX UIDEXCL file.

    DIRPOSIX UIDEXCL allows one or more UIDs to be specified on each line. If more than one UID is listed, one or more blanks must separate each entry. If asterisk (*) is the first non-blank character, the whole line is treated as a comment. All invalid UIDs are ignored.
12. If SYSENTRIES is in effect, the following POSIX groups are added, if groups with these names are not already defined:

Group name	GID
system	0
staff	1
bin	2
sys	3
adm	4
mail	6
security	7
nobody	4294967294

In addition, the following POSIX users are added, if they are not already defined:

Userid	UID	Primary group
root	0	system
daemon	1	staff
bin	2	bin
sys	3	sys
adm	4	adm
nobody	4294967294	nobody
default	4924967295	DEFAULT

Each user added has the following attributes:

- A password of NOLOG, indicating that this userid is not permitted to log on to the system. If your installation needs to log on to one of these users, that user's directory entry must be updated with a real password.
- Default and maximum storage sizes of 32M.
- Privilege class G only.
- No virtual devices.

If a special userid already exists with UID and/or primary group defined, that information will not be changed. However, the UID and/or primary group that is not already defined will be updated to values indicated in the above table.

If the UID for a special userid is already in use, the UID will be assigned to the special user and a warning message will be issued that indicates which other userid has that UID already assigned to it.

Note: Some External Security Managers (ESMs) may not support all of the entries created by the SYSENTRIES option. If you have an ESM installed, refer to the ESM publications for possible restrictions.

13. If SYSENTRIES is in effect, the user DEFAULT is created. This may be a problem in certain environments, such as if a profile exists with the name DEFAULT and a user DEFAULT exists (just added). This can cause confusion when updating the directory for certain programs that have a restriction of objects not having the same name.
14. If ONLYSYS is specified, the standard system group definitions and the standard system users will be defined if they do not already exist. If a system group or user ID already exists, its definition is not changed by DIRPOSIX, even if the definition does not match the tables in usage note 12 on page 162.
15. The internal form of the SYSAFFIN directory control statement may affect the assignment of a UID or a primary group (or both). If one or more POSIXINFO statements exist for different systems due to SYSAFFIN statements, then a UID and/or a primary group is not assigned to that user.
16. DIRPOSIX LOGFILE is created or appended to when DIRPOSIX executes. The logfile is a running history of the changes that DIRPOSIX has made (informational messages) as well as warning messages (potential conflicts). The file mode for DIRPOSIX LOGFILE is *outfm* or = (the same file mode as the created directory output file).

Examples

1. To invoke DIRPOSIX for file SYSTEM1 DIRECT C to assign unique UIDs in the range 200 to 2147483647 (the default values) to the userids that do not have a UID assigned, and with the standard system groups and users defined (default option), enter the following:

```
dirposix system1 direct c
```

As a result of this invocation, SYSTEM1 $DIRECT C will be created and DIRPOSIX LOGFILE C will be created or added to.

2. To invoke DIRPOSIX for file USER DIRECT * to assign unique UIDs in the 1000 to 2000 range to the userids that do not have a UID assigned, and with the standard system groups and users defined (default option) and replacing the output file (if found), enter the following:

   ```
   dirposix (minuid 1000 maxuid 2000 replace
   ```

   As a result of this invocation, USER $DIRECT * will be created or replaced, and DIRPOSIX LOGFILE * will be created or added to. (* indicates the file mode where USER DIRECT was found.)

3. To invoke DIRPOSIX for file USER DIRECT B to assign unique UIDs in the 50 to 999 range (except 100, 342, 511, 1000, 1995, 2000, 4096, 8192, 16384, 500000, and 3000000, which are in DIRPOSIX UIDEXCL *) to the userids (except ESM, OPERATOR, *USER, DEFAULT, DONTUSE, and POSIX, which are in DIRPOSIX USEREXCL *), with a primary group name of ClassG and the group id being 512, not adding the standard system group and user definitions, and replacing the output file (if found), enter the following:

   ```
   dirposix user direct b = = h (minuid 50 maxuid 999
    prim classg 512 nosys rep
   ```

   As a result of this invocation, USER $DIRECT H will be created or replaced, and DIRPOSIX LOGFILE H will be created or added to.

   DIRPOSIX USEREXCL could look like:

   ```
   * this line is a comment
   ESM
   OPERATOR  *USER                                DEFAULT
   DONTUSE  POSIX
   ```

   DIRPOSIX UIDEXCL could look like:

   ```
   * this line is a comment
   100  342             511        1000
   1995
        2000
                 4096                                  8192
   * this line is another comment
   16384
     500000 3000000
   ```

Return Codes

Code	Meaning
1	Invalid UID value *minuid* specified on field MINUID.
2	Invalid UID value *maxuid* specified on field MAXUID.
3	Parameter keyword *parm* was specified more than once or it conflicts with a previously specified entry.
4	Invalid Group name *gname* specified on field PRIMARYGROUP.
5	Unknown parameter *parm* given.
6	Input File *infn inft infm* not found or disk not accessed.
7	Output file *outfn outft outfm* exists and REPLACE option was not used.
8	Filemode *outfm* is not accessed read/write.
9	File specification *infn inft infm* is invalid.

10 UID specified on the MINUID (*minuid*) is below the minimum allowed UID value of 10.

11 Input file can not equal output file.

12 Value for MAXUID is less than the value for MINUID. This is not allowed.

13 The range of UIDs is not sufficient to assign a unique UID to each user in the input directory.

15 DIRPOSIX $TEMPDIR is a reserved filename. This filename can not be used as the input or output file specification for DIRPOSIX.

16 Invalid command line specification. Please check the command syntax.

100 Userid *userid* appears in the 'system users list' but is in the directory as a PROFILE entry.

101 Group *gname* appears in the 'system groups list' and is also specified in the current directory source. The GID is different between the two specifications. The directory version of this group will be respected.

102 The group you specified as your 'primary group' was defined in the source directory as having a GID equal to *gid*. This value for the GID is being used.

103 The group you specified as your 'primary group' was not defined in the source directory. No GID was provided on the command line. The operation involving the assignment of a primary group to each user will not take place.

104 Attention: The group you specified as your 'primary group' (*pipe*) is also defined as a system group to be added. The GID listed on the primary group specification (*pipe*) conflicts with the GID from the system group list (*pipe*). The primary group GID is being changed to reflect the GID from the system group specification.

105 Attention: A profile *profile* was detected with the same name as a system user id. While profile and user entries with the same name are allowed this could pose problems for some directory maintenance products.

106 Attention: A primary group of DEFAULT was specified with an optional GID. The DEFAULT group is used as a system default when assigning group membership to users without a specific GID or group name specified on their POSIXINFO directory statements. While the use of this value as the primary group is valid it should be noted that a POSIXGROUP statement will not be added for this entry. The trailing GID you provided will be ignored.

201 DirPosix Operation Starting *date_and_time* Command Line Parms: *command_line_parms*

202 Altered User *userid* UID *uid*

203 Left User *userid* UID *uid*

204 Skipped User *userid* - - - - User found in the excluded list

205 Added Group *gname* GID *gid*

206 Added User *userid* UID *uid* GNAME *gname*

207 Altered User *userid uid gname*

208 Skipped User *userid* - - - - Multiple sysaffin for POSIXINFO detected

209 Operation complete without severe errors.

210 Operation complete but errors were encountered.

301 Attention: DirPosix was unable to locate a DIRPOSIX USEREXCL file. No users are being considered excluded from DirPosix processing.

302 DirPosix is using *fn ft fm* as the USER EXCLUDE file.

303 Attention: DirPosix was unable to locate a DIRPOSIX UIDEXCL file. No UIDs are being considered excluded from DirPosix processing.

304 DirPosix is using *fn ft fm* as the UID EXCLUDE file.

305 DirPosix processing starting. Depending on the size of your source directory this could take awhile.

306 Attention: userid *userid* currently has a UID of *uid*. This is in conflict with a system user. The system user will be added but beware of the ramifications of users sharing UID values.

3209 Unexpected return code *rc* from pipe *pipe*

3210 Unexpected return code *rc* from command: *command_line*

3408 While processing directory entry *userid*, an error was detected while attempting to process a POSIXINFO quoted string. Specifically, two quoted strings were found to be adjacent to each other. For example:
'aaaaaa'"bbbbb"

This is not allowed.

Appendix B. OpenExtensions Shell Command Summary

The following list presents OpenExtensions shell commands and utilities grouped by the task a user might want to perform. Similar tasks are organized together. Commands that are OpenExtensions extensions to POSIX.2 are indicated with an "OE."

General Use

command	Run a simple command
cms	Invoke VM commands from the shell
date	Display the date and time
echo	Write arguments to standard output
print	Return arguments from the shell
printf	Write formatted output
sh	Invoke a shell (command interpreter)[1]
time	Display processor and elapsed times for a command
whence	Tell how the shell interprets a command name

Controlling Your Environment

alias	Display or create a command alias
env	Display environments, or set an environment for a process
export	Set the export attributes for variables, or show currently exported variables
fc	Process a command history list
id	Return the user identity
locale	Get locale-specific information
logger	Log messages
logname	Return a user's login name
newgrp	Change to a new group
readonly	Mark a variable as read-only
return	Return from a shell function or **.** (dot) script
set	Set or unset command options and positional parameters
shift	Shift positional parameters
stty	Set or display terminal options
su	Start a shell that runs with superuser privileges
touch	Change the file access and modification times
tty	Return the user's terminal name
unalias	Remove alias definitions
uname	Display the name of the current operating system
unset	Unset values and attributes of variables and functions

Managing Directories

basename	Return the nondirectory components of a path name
cd	Change the working directory
dirname	Return the directory components of a path name
ls	List file and directory names and attributes
mkdir	Make a directory

1. Use **OPENVM SHELL** to invoke the OpenExtensions shell initially.

mv	Rename or move a file or directory
pathchk	Check a path name
pwd	Return the working directory name
rm	Remove a directory entry
rmdir	Remove a directory

Managing Files

cat	Concatenate or display a text file
chgrp	Change the group owner of a file or directory
chmod	Change the mode of a group or directory
chown	Change the owner or group of a file or directory
cksum	Calculate and write checksums and byte counts
cmp	Compare two files
comm	Show and select or reject lines common to two files
cp	Copy a file
cut	Cut out selected fields of each line of a file
dd	Convert and copy a file
diff	Compare two text files and show the differences
ed	Use the **ed** line-oriented text editor
find	Find a file meeting specified criteria
fold	Break lines into shorter lines
head	Display the first part of a file
iconv	Convert characters from one code set to another
join	Join two sorted, textual relational databases
ln	Create a link to a file
mkfifo	Make a FIFO special file
mknod OE	Make a FIFO or character special file
od	Dump a file in a specified format
paste	Merge corresponding or subsequent lines of a file
sed	Start the **sed** noninteractive stream editor
sort	Start the sort-merge utility
tail	Display the last part of a file
tee	Duplicate the output stream
tr	Translate characters
umask	Set or return the file mode creation mask
uniq	Report or filter out repeated lines in a file
wc	Count newlines, words, and bytes

Printing Files

lp	Send a file to a printer
pr	Format a file in paginated form and send it to standard output

Computing and Managing Logic

bc	Use the arbitrary-precision arithmetic calculation language
break	Exit from a for, while, or until loop in a shell script
colon or **:**	Do nothing, successfully
continue	Skip to the next iteration of a loop in a shell script
dot or **.**	Run a shell file in the current environment
eval	Construct a command by concatenating arguments
exec	Run a command and open, close, or copy the file descriptors
exit	Return to the parent process from which the shell was called or to CMS
expr	Evaluate arguments as an expression
false	Return a nonzero exit code

grep	Search a file for a specified pattern
let	Evaluate an arithmetic expression
test	Test for a condition
trap	Intercept abnormal conditions and interrupts
true	Return a value of `0`

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
nohup	Start a process that is immune to hangups
ps	Return the status of a process
sleep	Suspend execution of a process for an interval of time
time	Display processor and elapsed times for a command
wait	Wait for a child process to end

Writing Shell Scripts

getconf	Get configuration values
getopts	Parse utility options
read	Read a line from standard input
type	Tell how the shell interprets a name
typeset	Assign attributes and values to variables
xargs	Construct an argument list and run a command

Developing or Porting Application Programs

ar	Create or maintain library archives
awk	Process programs written in the **awk** language
c89 or **cxx**	Compile C or C++ source code and create an executable file
lex	Generate a program for lexical tasks
make	Maintain program-generated and interdependent files
strip	Remove unnecessary information from an executable file
yacc	Use the **yacc** compiler

Communicating with the System or Other Users

mailx	Send or receive electronic mail

Working with Archives

cpio	Copy in/out file archives
pax	Interchange portable archives
tar	Manipulate the **tar** archive files to copy or back up a file

Appendix C. Using awk

PI

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**. Experienced programmers may prefer to turn directly to **awk** in *z/VM: OpenExtensions Commands 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:

```
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 11. The hobbies File

This file is included with the OpenExtensions shell as `/etc/samples/hobbies`.

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 processes 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 XEDIT editor to write an **awk** program and want to use horizontal tabs, see "Typing Tabs using XEDIT" on page 134.

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 quotation marks. 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 quotation marks. 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 quotation marks or apostrophe (') characters. The *datafile* operand gives the name of the data file. For example:

```
awk '$1 == "Linda"' hobbies
```

processes the program:

```
$1 == "Linda"
```

on the data file `hobbies`.

If you are using the OpenExtensions shell, you can type in a multiline program within single quotation marks, 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 lets you define the value of variables on the command line by using the **–v** option. See *z/VM: OpenExtensions Commands 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 process 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 *z/VM: OpenExtensions Commands 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 *z/VM: OpenExtensions Commands 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`). Because **.** 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, `*` usually 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 192.

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 FILE NAME ": " $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 file name, 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 separates 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 *z/VM: OpenExtensions Commands 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** processes 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 operand 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 operand *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(operand-list) {
    statements
}
```

The *operand-list* is a list of one or more names (separated by commas) that represent operand values passed to the function. When an operand name is used in the *statements* of a function, it is replaced by a copy of the corresponding operand value.

For example, the following is a simple function that takes a single numeric operand *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 operand 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 operand.

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 usual? 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:

- *Usual 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 193.

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.dddddd E+dd`.

f A floating-point number in conventional form, as in `-ddd.dddddd`.

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*
`\"`	Quotation Marks
`\`*c*	Any other character *c*


PI
end

Appendix D. The Format of Archive Files: cpio and tar

PI

You can use the **cpio** or **tar** command to back up or restore files. The **cpio** command reads and writes either a compact binary format header or an ASCII format header. The **tar** command reads and writes headers in either the original TAR format from UNIX systems or the USTAR format defined by the POSIX 1003.1 standard.

The **pax** command reads and writes headers in any of the **cpio** or **tar** formats.

This appendix describes the formats of the **cpio** and **tar** archive files.

cpio Format

A **cpio** archive consists of one or more concatenated member files. Each member file contains a header optionally followed by file contents as indicated in the header. The end of the archive is indicated by another header describing a file named **TRAILER!!!** which is empty.

There are two types of **cpio** archives, differing only in the style of the header:

- ASCII archives have totally printable header information; thus, if the files being archived are also ASCII files, the whole archive is ASCII.
- By default, **cpio** writes archives with binary headers; however, binary archive files cannot usually be ported to other operating systems, so it is recommended that you *not* use these.

The information in an ASCII archive header is stored in fixed-width, octal (base 8) numbers zero-padded on the left. The following table gives the order and field width for the information in the ASCII header:

Table 4. cpio Archive File: ASCII Header

Field Width	Field Name	Meaning
6	magic	Magic number 070707
6	dev	Device where file resides
6	ino	I-number of file
6	mode	File mode
6	uid	Owner user ID
6	gid	Owner group ID
6	nlink	Number of links to file
6	rdev	Device major/minor for special file
11	mtime	Modify time of file
6	namesize	Length of file name
11	filesize	Length of file

After the header information, *namesize* bytes of path name is stored. *namesize* includes the null byte of the end of the path name. After this, *filesize* bytes of the file contents are recorded.

Binary headers contain the same information in 2-byte (short) and 4-byte (long) integers as follows:

Table 5. cpio Archive File: Binary Header

Bytes	Field Name
2	magic
2	dev
2	ino
2	mode
2	uid
2	gid
2	nlink
2	rdev
4	mtime
2	namesize
2	reserved
4	filesize

After the header information comes the file name, with *namesize* rounded up to the nearest 2-byte boundary. Then the file contents appear as in the ASCII archive. The byte ordering of the 2- and 4-byte integers in the binary format is machine-dependent and thus portability of this format is not easily guaranteed.

Compressed **cpio** archives are exactly equivalent to the corresponding archive being passed to a 14-bit **compress** utility.

tar Format

The OpenExtensions **tar** utility supports both the older UNIX-compatible **tar** formats and the new USTAR format. The USTAR format allows more information to be stored and supports longer path names.

A **tar** archive, in either format, consists of one or more blocks, which represents member files. Each block is 512 bytes long; you can use the **-b** option with **tar** to indicate how many of these blocks are read or written (or both) at the same time.

Each member file consists of a header block, followed by zero or more blocks containing the file contents. The end of the archive is indicated by two blocks filled with binary zeros. Unused space in the header is left as binary zeros.

The header information in a block is stored in a printable ASCII form, so that **tar** archives are easily ported to different environments. If the contents of the files on the archive are all ASCII, the entire archive is ASCII.

Table 6 shows the UNIX format of the header block for a file.

Table 6. tar Archive File: UNIX-Compatible Format

Field Width	Field Name	Meaning
100	name	Name of file
8	mode	File mode
8	uid	Owner user ID
8	gid	Owner group ID
12	size	Length of file in bytes
12	mtime	Modify time of file
8	chksum	Checksum for header
1	link	Indicator for links
100	linkname	Name of linked file

- A directory is indicated by a trailing / (slash) in its name.
- The *link* field is:
  - 1 for a linked file
  - 2 for a symbolic link
  - 0 otherwise.

tar determines that the USTAR format is being used by the presence of the null-ended string `ustar` in the *magic* field. All fields before the *magic* field correspond to those of the UNIX format, except that *typeflag* replaces the link field.

Table 7. tar Archive File: USTAR Format

Field Width	Field Name	Meaning
100	name	Name of file
8	mode	File mode
8	uid	Owner user ID
8	gid	Owner group ID
12	size	Length of file in bytes
12	mtime	Modify time of file
8	chksum	Checksum for header
1	typeflag	Type of file
100	linkname	Name of linked file
6	magic	USTAR indicator
2	version	USTAR version
32	uname	Owner user name
32	gname	Owner group name
8	devmajor	Device major number
8	devminor	Device minor number
155	prefix	Prefix for file name

Description of the Header Fields

In the headers:

- The *name* field contains the name of the archived file. On USTAR format archives, the value of the prefix field, if non-null, is prefix to the *name* field to allow names longer than 100 characters.
- The *magic*, *uname*, and *gname* fields are null-ended character strings
- The *name*, *linkname*, and *prefix* fields are null-ended unless the full field stores a name (that is, the last character is not null).
- All other fields are zero-filled octal numbers, in ASCII. Trailing nulls are present for these numbers, except for the *size*, *mtime*, and *version* fields.
- *prefix* is null unless the file name exceeds 100 characters.
- The *size* field is zero if the header describes a link.
- The *chksum* field is a checksum of all the bytes in the header, assuming that the chksum field itself is all blanks.
- For USTAR, the *typeflag* field is a compatible extension of the *link* field of the older **tar** format. The following values are recognized:

Flag	File Type
0 or null	Regular file
1	Link to another file already archived
2	Symbolic link

3	Character special file
4	Block special file (not supported for OpenExtensions)
5	Directory
6	FIFO special file
7	Reserved
F	CMSDATA external link
G	CMSEXEC external link
H	MOUNT external link
A–Z	Available for custom usage (except for F, G and H that are used by OpenExtensions).

- In USTAR format, the *uname* and *gname* fields contain the name of the owner and group of the file, respectively.

Compressed **tar** archives are exactly equivalent to the corresponding archive being passed to a 14-bit **compress** utility.

Appendix E. Code Pages and the POSIX Portable Character Set

For purposes of comparison, this appendix includes:

- Latin 1/Open System Interconnection Code Page 01047 (IBM-1047)
- POSIX Portable Character Set
- U.S. APL Code Page 00293

Latin 1/Open System Interconnection Code Page 01047 (IBM-1047)

HEX DIGITS 1ST → 2ND ↓	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0	(SP) SP010000	& SM030000	- SP100000	ø LO610000	Ø LO620000	° SM190000	µ SM170000	¬ SM660000	{ SM110000	} SM140000	\ SM070000	0 ND100000
-1	(RSP) SP300000	é LE110000	/ SP120000	É LE120000	a LA010000	j LJ010000	~ SD190000	£ SC020000	A LA020000	J LJ020000	÷ SA060000	1 ND010000
-2	â LA150000	ê LE150000	Â LA160000	Ê LE160000	b LB010000	k LK010000	s LS010000	¥ SC050000	B LB020000	K LK020000	S LS020000	2 ND020000
-3	ä LA170000	ë LE170000	Ä LA180000	Ë LE180000	c LC010000	l LL010000	t LT010000	• SD630000	C LC020000	L LL020000	T LT020000	3 ND030000
-4	à LA130000	è LE130000	À LA140000	È LE140000	d LD010000	m LM010000	u LU010000	© SM520000	D LD020000	M LM020000	U LU020000	4 ND040000
-5	á LA110000	í LI110000	Á LA120000	Í LI120000	e LE010000	n LN010000	v LV010000	§ SM240000	E LE020000	N LN020000	V LV020000	5 ND050000
-6	ã LA190000	î LI150000	Ã LA200000	Î LI160000	f LF010000	o LO010000	w LW010000	¶ SM250000	F LF020000	O LO020000	W LW020000	6 ND060000
-7	å LA270000	ï LI170000	Å LA280000	Ï LI180000	g LG010000	p LP010000	x LX010000	¼ NF040000	G LG020000	P LP020000	X LX020000	7 ND070000
-8	ç LC410000	ì LI130000	Ç LC420000	Ì LI140000	h LH010000	q LQ010000	y LY010000	½ NF010000	H LH020000	Q LQ020000	Y LY020000	8 ND080000
-9	ñ LN190000	ß LS610000	Ñ LN200000	` SD130000	i LI010000	r LR010000	z LZ010000	¾ NF050000	I LI020000	R LR020000	Z LZ020000	9 ND090000
-A	¢ SC040000	! SP020000	¦ SM650000	: SP130000	« SP170000	ª SM210000	¡ SP030000	Ý LY120000	(SHY) SP320000	1 ND011000	2 ND021000	3 ND031000
-B	. SP110000	$ SC030000	, SP080000	# SM010000	» SP180000	º SM200000	¿ SP160000	¨ SD170000	ô LO150000	û LU150000	Ô LO160000	Û LU160000
-C	< SA030000	* SM040000	% SM020000	@ SM050000	ð LD630000	æ LA510000	Ð LD620000	¯ SM150000	ö LO170000	ü LU170000	Ö LO180000	Ü LU180000
-D	(SP060000	) SP070000	_ SP090000	' SP050000	ý LY110000	¸ SD410000	[SM060000	] SM080000	ò LO130000	ù LU130000	Ò LO140000	Ù LU140000
-E	+ SA010000	; SP140000	> SA050000	= SA040000	þ LT630000	Æ LA520000	Þ LT640000	´ SD110000	ó LO110000	ú LU110000	Ó LO120000	Ú LU120000
-F	| SM130000	^ SD150000	? SP150000	" SP040000	± SA020000	¤ SC010000	® SM530000	× SA070000	õ LO190000	ÿ LY170000	Õ LO200000	(EO)

Code Page 01047

Figure 12. Latin 1/Open System Interconnection Code Page 01047 (IBM-1047)

POSIX Portable Character Set 00103

A LA020000	B LB020000	C LC020000	D LD020000	E LE020000	F LF020000	G LG020000	H LH020000	I LI020000	J LJ020000	K LK020000	L LL020000	M LM020000
N LN020000	O LO020000	P LP020000	Q LQ020000	R LR020000	S LS020000	T LT020000	U LU020000	V LV020000	W LW020000	X LX020000	Y LY020000	Z LZ020000
a LA010000	b LB010000	c LC010000	d LD010000	e LE010000	f LF010000	g LG010000	h LH010000	i LI010000	j LJ010000	k LK010000	l LL010000	m LM010000
n LN010000	o LO010000	p LP010000	q LQ010000	r LR010000	s LS010000	t LT010000	u LU010000	v LV010000	w LW010000	x LX010000	y LY010000	z LZ010000
0 ND100000	1 ND010000	2 ND020000	3 ND030000	4 ND040000	5 ND050000	6 ND060000	7 ND070000	8 ND080000	9 ND090000			
+ SA010000	< SA030000	= SA040000	> SA050000	$ SC030000	` SD130000	^ SD150000	~ SD190000	# SM010000	% SM020000	& SM030000	* SM040000	@ SM050000
[SM060000	\ SM070000	] SM080000	{ SM110000	| SM130000	} SM140000	! SP020000	" SP040000	' SP050000	(SP060000	) SP070000	, SP080000	_ SP090000
- SP100000	. SP110000	/ SP120000	: SP130000	; SP140000	? SP150000							

Character Set 00103

Figure 13. POSIX Portable Character Set 00103

U.S. APL Code Page 00293

HEX DIGITS 1ST → 2ND ↓	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0	(SP) SP010000	& SM030000	– SL690000	◇ SL370000	~ SL460000	□ SL360000	¯ SL630000	α SL710000	{ SM110000	} SM140000	\ SM070000	0 ND100000
-1	A LA480000	J LJ480000	/ SL760000	∧ SL510000	a LA010000	j LJ010000	~ SD190000	∈ SL720000	A LA020000	J LJ020000	Ξ SL300000	1 ND010000
-2	B LB480000	K LK480000	S LS480000	¨ SL450000	b LB010000	k LK010000	s LS010000	⍳ SL730000	B LB020000	K LK020000	S LS020000	2 ND020000
-3	C LC480000	L LL480000	T LT480000	⍰ SL270000	c LC010000	l LL010000	t LT010000	ρ SL740000	C LC020000	L LL020000	T LT020000	3 ND030000
-4	D LD480000	M LM480000	U LU480000	⍳ SL860000	d LD010000	m LM010000	u LU010000	ω SL750000	D LD020000	M LM020000	U LU020000	4 ND040000
-5	E LE480000	N LN480000	V LV480000	∈ SL870000	e LE010000	n LN010000	v LV010000		E LE020000	N LN020000	V LV020000	5 ND050000
-6	F LF480000	O LO480000	W LW480000	⊢ SL340000	f LF010000	o LO010000	w LW010000	× SL550000	F LF020000	O LO020000	W LW020000	6 ND060000
-7	G LG480000	P LP480000	X LX480000	⊣ SL350000	g LG010000	p LP010000	x LX010000	\ SL640000	G LG020000	P LP020000	X LX020000	7 ND070000
-8	H LH480000	Q LQ480000	Y LY480000	∨ SL500000	h LH010000	q LQ010000	y LY010000	÷ SL540000	H LH020000	Q LQ020000	Y LY020000	8 ND080000
-9	I LI480000	R LR480000	Z LZ480000	` SD130000	i LI010000	r LR010000	z LZ010000		I LI020000	R LR020000	Z LZ020000	9 ND090000
-A	¢ SC040000	! SP020000	¦ SM650000	: SL830000	↑ SL610000	⊃ SL430000	∩ SL400000	∇ SL030000	⍲ SL170000	⌶ SL240000	≠ SL150000	
-B	. SL840000	$ SC030000	, SL850000	# SM010000	↓ SL620000	⊂ SL420000	∪ SL410000	∆ SL060000	⍱ SL180000	! SL580000	⍀ SL160000	⍒ SL040000
-C	< SL520000	* SL650000	% SM020000	@ SM050000	≤ SL560000		⊥ SL230000	⊤ SL220000	▯ SL260000	⍧ SL050000	⁖ SL320000	⍋ SL330000
-D	(SL670000	) SL680000	_ SL440000	' SL660000	⌈ SL010000	○ SL080000	[SL770000	] SL780000	⌽ SL090000	⍋ SL070000	⊖ SL120000	⍟ SL110000
-E	+ SL790000	; SL800000	> SL530000	= SL810000	⌊ SL020000		≥ SL570000	≠ SL820000	⍂ SL280000	⍞ SL130000	⍠ SL140000	⍎ SL190000
-F	| SM130000	¬ SM660000	? SL700000	" SP040000	→ SL600000	← SL590000	∘ SL250000	| SL380000	⍉ SL100000	⍵ SL210000	⍕ SL200000	(EO)

Code Page 00293

Figure 14. U.S. APL Code Page 00293

Appendix F. Escape Sequences

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 204.

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. In Table 8, the default escape character, the cent sign (¢), is used.

Table 8. Portable Characters: Escape Sequences

Portable Character	ASCII Control Sequence	OpenExtensions Escape Sequence
<NUL>	control-@	¢@ or ¢0
<alert>	control-G	¢g or ¢G
<backspace>	control-H	¢h or ¢H
<tab>	control-I	¢i or ¢I
<newline>	control-J	¢j or ¢J
<vertical-tab>	control-K	¢k or ¢K
<form-feed>	control-L	¢l or ¢L
<carriage-return>	control-M	¢m or ¢M
<left-square-bracket>	left-square-bracket	¢(or ¢{
<right-square-bracket>	right-square-bracket	¢) or ¢}

<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 XEDIT editor, you cannot type a <tab> character (XEDIT handles only displayable characters). See “Typing Tabs using XEDIT” on page 134 for information on how to enter a <tab> character when using the XEDIT.

Escape Sequences for Control Characters

To obtain the following control characters, you must use an escape sequence. In Table 9, the default escape character, the cent sign (¢), is used.

Table 9. Control Characters: Escape Sequences

Control Character	ASCII Control Sequence	OpenExtensions Escape Sequence
<ACK>	control-F	¢f or ¢F
<CAN>	control-X	¢x or ¢X
<DC1>	control-Q	¢q or ¢Q
<DC2>	control-R	¢r or ¢R
<DC3>	control-S	¢s or ¢S
<DC4>	control-T	¢t or ¢T
<DEL>	control-?	¢?, ¢7, or ¢#
<DLE>	control-P	¢p or ¢P
<EM>	control-Y	¢y or ¢Y
<ENQ>	control-E	¢e or ¢E
<EOT>	control-D	¢d or ¢D
<ESC>	control-left-square-bracket	¢2 or ¢-left-square-bracket
<ETB>	control-W	¢w or ¢W
<ETX>	control-C	¢c or ¢C
<IS1>	control-_	¢6 or ¢_
<IS2>	control-circumflex	¢5 or ¢¬
<IS3>	control-right-square-bracket	¢4 or ¢-right-square-bracket
<IS4>	control-backslash	¢3 or ¢backslash
<NAK>	control-U	¢u or ¢U
<SI>	control-O	¢o or ¢O
<SO>	control-N	¢n or ¢N
<SOH>	control-A	¢a or ¢A
<STX>	control-B	¢b or ¢B
<SUB>	control-Z	¢z or ¢Z
<SYN>	control-V	¢v or ¢V

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 all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, New York 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs

and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
2455 South Road
Poughkeepsie, New York 12601-5400
U.S.A.
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 document 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.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information may contain sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Programming Interface Information

This publication primarily documents information that is NOT intended to be used as Programming Interfaces of z/VM.

This publication also documents intended Programming Interfaces that allow the customer to write programs to obtain the services of z/VM. This information is identified where it occurs, either by an introductory statement to a chapter or section or by the following marking:

PI

<...Programming Interface information...>

PI end

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml

Adobe is either a registered trademark or a trademark of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Glossary

For a list of z/VM terms and their definitions, see *z/VM: Glossary*.

The z/VM glossary is also available through the online z/VM HELP Facility. For example, to display the definition of the term “dedicated device”, issue the following HELP command:

```
help glossary dedicated device
```

While you are in the glossary help file, you can do additional searches:

- To display the definition of a new term, type a new HELP command on the command line:

  ```
  help glossary newterm
  ```

  This command opens a new help file inside the previous help file. You can repeat this process many times. The status area in the lower right corner of the screen shows how many help files you have open. To close the current file, press the Quit key (PF3/F3). To exit from the HELP Facility, press the Return key (PF4/F4).
- To search for a word, phrase, or character string, type it on the command line and press the Clocate key (PF5/F5). To find other occurrences, press the key multiple times.

  The Clocate function searches from the current location to the end of the file. It does not wrap. To search the whole file, press the Top key (PF2/F2) to go to the top of the file before using Clocate.

Bibliography

See the following publications for additional information about z/VM. For abstracts of the z/VM publications, see *z/VM: General Information*.

Where to Get z/VM Information

z/VM product information is available from the following sources:

- z/VM Information Center at publib.boulder.ibm.com/infocenter/zvm/v6r1/index.jsp
- z/VM Internet Library at www.ibm.com/eserver/zseries/zvm/library/
- IBM Publications Center at www.elink.ibmlink.ibm.com/publications/servlet/pbi.wss
- *IBM Online Library: z/VM Collection on DVD*, SK5T-7054

z/VM Base Library

Overview

- *z/VM: General Information*, GC24-6193
- *z/VM: Glossary*, GC24-6195
- *z/VM: License Information*, GC24-6200

Installation, Migration, and Service

- *z/VM: Guide for Automated Installation and Service*, GC24-6197
- *z/VM: Migration Guide*, GC24-6201
- *z/VM: Service Guide*, GC24-6232
- *z/VM: VMSES/E Introduction and Reference*, GC24-6243

Planning and Administration

- *z/VM: CMS File Pool Planning, Administration, and Operation*, SC24-6167
- *z/VM: CMS Planning and Administration*, SC24-6171
- *z/VM: Connectivity*, SC24-6174
- *z/VM: CP Planning and Administration*, SC24-6178
- *z/VM: Getting Started with Linux on System z*, SC24-6194
- *z/VM: Group Control System*, SC24-6196
- *z/VM: I/O Configuration*, SC24-6198
- *z/VM: Running Guest Operating Systems*, SC24-6228
- *z/VM: Saved Segments Planning and Administration*, SC24-6229
- *z/VM: Secure Configuration Guide*, SC24-6230
- *z/VM: TCP/IP LDAP Administration Guide*, SC24-6236
- *z/VM: TCP/IP Planning and Customization*, SC24-6238
- *z/OS and z/VM: Hardware Configuration Manager User's Guide*, SC33-7989

Customization and Tuning

- *z/VM: CP Exit Customization*, SC24-6176
- *z/VM: Performance*, SC24-6208

Operation and Use

- *z/VM: CMS Commands and Utilities Reference*, SC24-6166
- *z/VM: CMS Pipelines Reference*, SC24-6169
- *z/VM: CMS Pipelines User's Guide*, SC24-6170
- *z/VM: CMS Primer*, SC24-6172
- *z/VM: CMS User's Guide*, SC24-6173
- *z/VM: CP Commands and Utilities Reference*, SC24-6175
- *z/VM: System Operation*, SC24-6233
- *z/VM: TCP/IP User's Guide*, SC24-6240
- *z/VM: Virtual Machine Operation*, SC24-6241
- *z/VM: XEDIT Commands and Macros Reference*, SC24-6244
- *z/VM: XEDIT User's Guide*, SC24-6245
- *CMS/TSO Pipelines: Author's Edition*, SL26-0018

Application Programming

- *z/VM: CMS Application Development Guide*, SC24-6162
- *z/VM: CMS Application Development Guide for Assembler*, SC24-6163
- *z/VM: CMS Application Multitasking*, SC24-6164
- *z/VM: CMS Callable Services Reference*, SC24-6165
- *z/VM: CMS Macros and Functions Reference*, SC24-6168

- *z/VM: CP Programming Services*, SC24-6179
- *z/VM: CPI Communications User's Guide*, SC24-6180
- *z/VM: Enterprise Systems Architecture/ Extended Configuration Principles of Operation*, SC24-6192
- *z/VM: Language Environment User's Guide*, SC24-6199
- *z/VM: OpenExtensions Advanced Application Programming Tools*, SC24-6202
- *z/VM: OpenExtensions Callable Services Reference*, SC24-6203
- *z/VM: OpenExtensions Commands Reference*, SC24-6204
- *z/VM: OpenExtensions POSIX Conformance Document*, GC24-6205
- *z/VM: OpenExtensions User's Guide*, SC24-6206
- *z/VM: Program Management Binder for CMS*, SC24-6211
- *z/VM: Reusable Server Kernel Programmer's Guide and Reference*, SC24-6220
- *z/VM: REXX/VM Reference*, SC24-6221
- *z/VM: REXX/VM User's Guide*, SC24-6222
- *z/VM: Systems Management Application Programming*, SC24-6234
- *z/VM: TCP/IP Programmer's Reference*, SC24-6239
- *Common Programming Interface Communications Reference*, SC26-4399
- *Common Programming Interface Resource Recovery Reference*, SC31-6821
- *z/OS: IBM Tivoli Directory Server Plug-in Reference for z/OS*, SA76-0148
- *z/OS: Language Environment Concepts Guide*, SA22-7567
- *z/OS: Language Environment Debugging Guide*, GA22-7560
- *z/OS: Language Environment Programming Guide*, SA22-7561
- *z/OS: Language Environment Programming Reference*, SA22-7562
- *z/OS: Language Environment Run-Time Messages*, SA22-7566
- *z/OS: Language Environment Writing ILC Applications*, SA22-7563
- *z/OS MVS Program Management: Advanced Facilities*, SA22-7644
- *z/OS MVS Program Management: User's Guide and Reference*, SA22-7643

Diagnosis

- *z/VM: CMS and REXX/VM Messages and Codes*, GC24-6161
- *z/VM: CP Messages and Codes*, GC24-6177
- *z/VM: Diagnosis Guide*, GC24-6187
- *z/VM: Dump Viewing Facility*, GC24-6191
- *z/VM: Other Components Messages and Codes*, GC24-6207
- *z/VM: TCP/IP Diagnosis Guide*, GC24-6235
- *z/VM: TCP/IP Messages and Codes*, GC24-6237
- *z/VM: VM Dump Tool*, GC24-6242
- *z/OS and z/VM: Hardware Configuration Definition Messages*, SC33-7986

z/VM Facilities and Features

Data Facility Storage Management Subsystem for VM

- *z/VM: DFSMS/VM Customization*, SC24-6181
- *z/VM: DFSMS/VM Diagnosis Guide*, GC24-6182
- *z/VM: DFSMS/VM Messages and Codes*, GC24-6183
- *z/VM: DFSMS/VM Planning Guide*, SC24-6184
- *z/VM: DFSMS/VM Removable Media Services*, SC24-6185
- *z/VM: DFSMS/VM Storage Administration*, SC24-6186

Directory Maintenance Facility for z/VM

- *z/VM: Directory Maintenance Facility Commands Reference*, SC24-6188
- *z/VM: Directory Maintenance Facility Messages*, GC24-6189
- *z/VM: Directory Maintenance Facility Tailoring and Administration Guide*, SC24-6190

Open Systems Adapter/Support Facility

- *System z10, System z9 and eServer zSeries: Open Systems Adapter-Express Customer's Guide and Reference*, SA22-7935
- *System z9 and eServer zSeries 890 and 990: Open Systems Adapter-Express Integrated Console Controller User's Guide*, SA22-7990

- *System z: Open Systems Adapter-Express Integrated Console Controller 3215 Support*, SA23-2247

Performance Toolkit for VM™

- *z/VM: Performance Toolkit Guide*, SC24-6209
- *z/VM: Performance Toolkit Reference*, SC24-6210

RACF® Security Server for z/VM

- *z/VM: RACF Security Server Auditor's Guide*, SC24-6212
- *z/VM: RACF Security Server Command Language Reference*, SC24-6213
- *z/VM: RACF Security Server Diagnosis Guide*, GC24-6214
- *z/VM: RACF Security Server General User's Guide*, SC24-6215
- *z/VM: RACF Security Server Macros and Interfaces*, SC24-6216
- *z/VM: RACF Security Server Messages and Codes*, GC24-6217
- *z/VM: RACF Security Server Security Administrator's Guide*, SC24-6218
- *z/VM: RACF Security Server System Programmer's Guide*, SC24-6219
- *z/VM: Security Server RACROUTE Macro Reference*, SC24-6231

Remote Spooling Communications Subsystem Networking for z/VM

- *z/VM: RSCS Networking Diagnosis*, GC24-6223
- *z/VM: RSCS Networking Exit Customization*, SC24-6224
- *z/VM: RSCS Networking Messages and Codes*, GC24-6225
- *z/VM: RSCS Networking Operation and Use*, SC24-6226
- *z/VM: RSCS Networking Planning and Configuration*, SC24-6227
- *Network Job Entry: Formats and Protocols*, SA22-7539

Prerequisite Products

Device Support Facilities

- *Device Support Facilities: User's Guide and Reference*, GC35-0033

Environmental Record Editing and Printing Program

- *Environmental Record Editing and Printing Program (EREP): Reference*, GC35-0152
- *Environmental Record Editing and Printing Program (EREP): User's Guide*, GC35-0151

Additional Publications

XL C/C++ for z/VM: Runtime Library Reference, SC09-7624

XL C/C++ for z/VM: User's Guide, SC09-7625

z/OS: XL C/C++ Programming Guide, SC09-4765-08

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

Program Number: 5741-A07

Printed in USA

SC24-6206-00