z/OS Version 2 Release 3

UNIX System Services User's Guide



Note

Before using this information and the product it supports, read the information in <u>"Notices" on page</u> 321.

This edition applies to Version 2 Release 3 of z/OS (5650-ZOS) and to all subsequent releases and modifications until otherwise indicated in new editions.

Last updated: 2019-02-16

[©] Copyright International Business Machines Corporation 1996, 2018.

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

Contents

List of Tables xvii About this document xix Who should use z/OS UNIX System Services User's Guide? xix What is in z/OS UNIX System Services User's Guide? xix Tasks that can be performed in more than one environment. xix z/OS information	List of Figures	xv
About this documentxix Who should use z/OS UNIX System Services User's Guide?xix Tasks that can be performed in more than one environmentxix ZOS Information	List of Tables	xvii
About this document		
About this document Xix Who should use z/OS UNIX System Services User's Guide? xix What is in z/OS UNIX System Services User's Guide? xix Tasks that can be performed in more than one environment. xix z/OS information. xix How to send your comments to IBM. xix Summary of changes xxii Summary of changes for z/OS UNIX for Version 2 Release 3 (V2R3). xxii Summary of changes for z/OS UNIX for Version 2 Release 3 (V2R2). xxii Z/OS Version 2 Release 1 summary of changes. xxii Part 1. The z/OS shells. 1 Chapter 1. An introduction to the z/OS shells. 3 About shells. 3 The locale in the shells. 4 Daemon support. 4 Running an X-Window application. 4 Accessing the shells – the choices. 5 Terminal emulators. 5 Interoperability between the shells and MVS. 6 Parallels between the MVS environment and the shell environment. 7 Programming for everyday tasks. 8 Editing. 9 Detaminal emulators. 11 <t< td=""><td>About this decument</td><td>viv</td></t<>	About this decument	viv
Who should use 2/05 UNLX System Services User's Guide? Xix What is in 2/05 UNLX System Services User's Guide? Xix Tasks that can be performed in more than one environment xix z/OS information xix How to send your comments to IBM xix Summary of changes for z/OS UNIX for Version 2 Release 3 (V2R3) xii Summary of changes for z/OS UNIX for Version 2 Release 3 (V2R3) xii Summary of changes for z/OS UNIX for Version 2 Release 2 (V2R2) xxii z/OS Version 2 Release 1 summary of changes. xxii Part 1. The z/OS shells. 3 About shells. 3 The locale in the shells. 4 Daemon support. 4 Running an X-Window application. 4 The shell user. 5 Terminal emulators. 5 Terminal emulators. 5 So control. 9 Dackground Jobs. 9 Data management. 9 Detarging options on the OMVS command. 11 Differences from a UNIX or AIX environment. 12 Understanding the shell screen. 12 Understanding the shell screen.	ADOUT THIS document	•••••• XIX
What is in 2/05 UNIX System Services Guide? Xix Tasks that can be performed in more than one environment. Xix z/05 information. xix How to send your comments to IBM. xix Summary of changes for z/05 UNIX for Version 2 Release 3 (V2R3). xxii Summary of changes for z/05 UNIX for Version 2 Release 3 (V2R2). xxii Z/05 Version 2 Release 1 summary of changes. xxii Part 1. The z/OS shells. 1 Chapter 1. An introduction to the z/OS shells. 3 About shells. 3 Shell commands and utilities. 3 The locale in the shells. 4 Daemon support. 4 Accessing the shells - the choices. 5 Terminal emulators. 5 Interoperability between the shells and MVS. 6 Paratlels between the MVS environment and the shell environment. 7 Programming for everyday tasks. 8 Editing. 9 Debugging. 9 Debugging. 9 Debugging. 9 Debugging. 9 Debugging options on the OMVS command. 12	Who should use Z/OS UNIX System Services User's Guide?	XIX
Tasks that can be performed in more than one environment. Xix z/OS information xix How to send your comments to IBM. xxi If you have a technical problem xxi Summary of changes for z/OS UNIX for Version 2 Release 3 (V2R3) xxii Summary of changes for z/OS UNIX for Version 2 Release 2 (V2R2) xxii z/OS Version 2 Release 1 summary of changes xxii Part 1. The z/OS shells 1 Chapter 1. An introduction to the z/OS shells 3 About shells 3 Shell commands and utilities 3 The locale in the shells 4 Daemon support 4 Running an X-Window application 4 Accessing the shells – the choices 5 Terminal emulators 5 Interoperability between the shells and MVS 6 Parallels between the MVS environment and the shell environment 7 Programming for everyday tasks 8 Editing 9 Data management 9 Debugging 9 Data management 12 Understanding the shell 12 Understa	What is in Z/US UNIX System Services User's Guide?	XIX
2/OS information Xix How to send your comments to IBM. Xix If you have a technical problem. xxi Summary of changes for z/OS UNIX for Version 2 Release 3 (V2R3) xxii Summary of changes for z/OS UNIX for Version 2 Release 3 (V2R3) xxii z/OS Version 2 Release 1 summary of changes xxii Part 1. The z/OS shells 1 Chapter 1. An introduction to the z/OS shells. 3 About shells 3 Shell commands and utilities. 3 The locale in the shells. 4 Daemon support 4 Running an X-Window application. 4 The shell user 4 Security. 4 Accessing the shells – the choices. 5 Terminal emulators. 5 Interoperability between the shells and MVS. 6 Paratlels between the shells and MVS. 6 Paratlels between the MVS environment and the shell environment. 7 Programming for everyday tasks. 8 Editing. 9 Debording. 9 Programming. 9 Debugging. 9 <	lasks that can be performed in more than one environment	XIX
How to Send your comments to IBM	Z/US Information	XIX
It you have a technical problem	How to send your comments to IBM	XXI
Summary of changes. XXII Summary of changes for z/OS UNIX for Version 2 Release 3 (V2R3). XXII Summary of changes for z/OS UNIX for Version 2 Release 2 (V2R2). XXII z/OS Version 2 Release 1 summary of changes. XXII Part 1. The z/OS shells. 1 Chapter 1. An introduction to the z/OS shells. 3 About shells. 3 Shell commands and utilities. 3 The locale in the shells. 4 Daemon support. 4 Running an X-Window application. 4 The shell user. 4 Security. 4 Accessing the shells – the choices. 5 Terminal emulators. 5 Interoperability between the shells and MVS. 6 Parallels between the MVS environment and the shell environment. 7 Programming for everyday tasks. 8 Editing. 9 Job control. 9 Background jobs. 9 Programming. 9 Debugging. 9 Data management. 11 Drifferences from a UNIX or AIX environment. 11	If you have a technical problem	XXI
Summary of changes for z/OS UNIX for Version 2 Release 3 (V2R3)	Summary of changes	XXİİ
Summary of changes for z/OS UNIX for Version 2 Release 2 (V2R2)	Summary of changes for z/OS UNIX for Version 2 Release 3 (V2R3)	xxii
z/OS Version 2 Release 1 summary of changes	Summary of changes for z/OS UNIX for Version 2 Release 2 (V2R2)	xxii
Part 1. The z/OS shells. 1 Chapter 1. An introduction to the z/OS shells. 3 About shells. 3 Shell commands and utilities. 3 The locale in the shells. 4 Daemon support. 4 Running an X-Window application. 4 The shell user. 4 Security. 4 Accessing the shells – the choices. 5 Terminal emulators. 5 Interoperability between the shells and MVS. 6 Parallels between the MVS environment and the shell environment. 7 Programming for everyday tasks. 8 Editing. 8 Job control. 9 Background jobs. 9 Programming. 9 Debugging. 9 Data management. 9 Ohapter 2. OMVS, a 3270 terminal interface to the z/OS shell. 11 Differences from a UNIX or AIX environment. 11 Invoking the shell screen. 12 Working in line mode. 12 Working in line mode. 14 Why isn't your output displayed on the screen? 1	z/OS Version 2 Release 1 summary of changes	xxii
Part 1. The z/OS shells. 1 Chapter 1. An introduction to the z/OS shells. 3 About shells. 3 Shell commands and utilities. 3 The locale in the shells. 4 Daemon support. 4 Running an X-Window application. 4 The shell user. 4 Security. 4 Accessing the shells – the choices. 5 Terminal emulators. 5 Interoperability between the shells and MVS. 6 Parallels between the MVS environment and the shell environment. 7 Programming for everyday tasks. 8 Editing. 8 Job control. 9 Background jobs. 9 Programming. 9 Data management. 9 Data management. 11 Differences from a UNIX or AIX environment. 11 Differences from a UNIX or AIX environment. 12 Working in line mode. 14 Why isn't your output displayed on the screen? 14 Why isn't your output displayed on the screen? 14 Determining function key sett		_
Chapter 1. An introduction to the z/OS shells. 3 About shells. 3 Shell commands and utilities. 3 The locale in the shells. 4 Daemon support. 4 Running an X-Window application. 4 The shell user. 4 Security. 4 Accessing the shells – the choices. 5 Terminal emulators. 5 Interoperability between the shells and MVS. 6 Parallels between the MVS environment and the shell environment. 7 Programming for everyday tasks. 8 Editing. 8 Job control. 9 Background jobs. 9 Programming. 9 Debugging. 9 Data management. 9 Chapter 2. OMVS, a 3270 terminal interface to the z/OS shell. 11 Differences from a UNIX or AIX environment. 12 Working the shell screen. 12 Working in line mode. 14 Why isn't your output displayed on the screen? 14 Determining function key settings and the escape character. 15 The	Part 1. The z/OS shells	1
Chapter 1. An introduction to the z/OS shells. 3 About shells. 3 Shell commands and utilities. 3 The locale in the shells. 4 Daemon support. 4 Running an X-Window application. 4 The shell user. 4 Security. 4 Accessing the shells – the choices. 5 Terminal emulators. 5 Interoperability between the shells and MVS. 6 Parallels between the MVS environment and the shell environment. 7 Programming for everyday tasks. 8 Editing. 8 Job control. 9 Background jobs. 9 Programming. 9 Debugging. 9 Debugging. 9 Data management. 9 Chapter 2. OMVS, a 3270 terminal interface to the z/OS shell. 11 Invoking the shell. 11 Differences from a UNIX or AIX environment. 11 Invoking in line mode. 12 Working in line mode. 14 Why isn't your outiput displayed on the screen? 14		
About shells 3 Shell commands and utilities. 3 The locale in the shells 4 Daemon support. 4 Running an X-Window application 4 The shell user. 4 Security. 4 Accessing the shells – the choices. 5 Terminal emulators. 5 Interoperability between the shells and MVS. 6 Parallels between the MVS environment and the shell environment. 7 Programming for everyday tasks. 8 Editing. 8 Job control. 9 Background jobs. 9 Programming. 9 Data management. 9 Differences from a UNIX or AIX environment. 11 Divoking the shell. 11 Differences from a UNIX or AIX environment. 12 Understanding the shell screen. 12 Working in line mode. 14 Why isn't your output displayed on the screen? 14 Determining function key settings and the escape character. 15 The function key functions. 15 The scape character	Chapter 1. An introduction to the z/OS shells	3
Shell commands and utilities	About shells	3
The locale in the shells. 4 Daemon support. 4 Running an X-Window application. 4 The shell user. 4 Security. 4 Accessing the shells – the choices. 5 Terminal emulators. 5 Interoperability between the shells and MVS. 6 Parallels between the MVS environment and the shell environment. 7 Programming for everyday tasks. 8 Editing. 8 Job control. 9 Background jobs. 9 Programming. 9 Debugging. 9 Data management. 9 Chapter 2. OMVS, a 3270 terminal interface to the z/OS shell. 11 Differences from a UNIX or AIX environment. 11 Invoking the shell. 11 Understanding the shell screen. 12 Working in line mode. 14 Why isn't your output displayed on the screen? 14 Why isn't your output displayed on the screen? 14 Determing function key settings and the escape character. 15 The scape character. 18 <td< td=""><td>Shell commands and utilities</td><td></td></td<>	Shell commands and utilities	
Daemon support. 4 Running an X-Window application. 4 The shell user. 4 Security. 4 Accessing the shells – the choices. 5 Terminal emulators. 5 Interoperability between the shells and MVS. 6 Parallels between the MVS environment and the shell environment. 7 Programming for everyday tasks. 8 Editing. 8 Job control. 9 Background jobs. 9 Programming. 9 Debugging. 9 Data management. 9 Chapter 2. OMVS, a 3270 terminal interface to the z/OS shell. 11 Differences from a UNIX or AIX environment. 11 Invoking the shell. 11 Understanding the shell screen. 12 Working in line mode. 14 Why isn't your output displayed on the screen? 14 Determining function key settings and the escape character. 15 The use the variant characters on your keyboard. 18 Entering a shell command. 18 Customizing the variant characters on your keyboard. <td< td=""><td>The locale in the shells</td><td></td></td<>	The locale in the shells	
Running an X-Window application. 4 The shell user. 4 Security. 4 Accessing the shells – the choices. 5 Terminal emulators. 5 Interoperability between the shells and MVS. 6 Parallels between the MVS environment and the shell environment. 7 Programming for everyday tasks. 8 Editing. 8 Job control. 9 Background jobs. 9 Programming. 9 Debugging. 9 Data management. 9 Chapter 2. OMVS, a 3270 terminal interface to the z/OS shell. 11 Differences from a UNIX or AIX environment. 11 Invoking the shell. 11 Understanding the shell screen. 12 Working in line mode. 12 Working in line mode. 14 Why isn't your output displayed on the screen? 14 Why isn't your output displayed on the screen? 14 Determing function key settings and the escape character. 15 The function key settings and the escape character. 15 The scape character. 18	Daemon support	
The shell user. 4 Security. 4 Accessing the shells – the choices. 5 Terminal emulators. 5 Interoperability between the shells and MVS. 6 Parallels between the MVS environment and the shell environment. 7 Programming for everyday tasks. 8 Editing. 8 Job control. 9 Background jobs. 9 Programming. 9 Debugging. 9 Data management. 9 Chapter 2. OMVS, a 3270 terminal interface to the z/OS shell. 11 Differences from a UNIX or AIX environment. 11 Invoking the shell. 11 Chapter 2. OMVS, a 3270 terminal interface to the z/OS shell. 11 Differences from a UNIX or AIX environment. 11 Invoking the shell. 11 Chapter 2. OMVS, a 3270 terminal interface to the z/OS shell. 11 Differences from a UNIX or AIX environment. 11 Invoking the shell. 11 Chapter 3. 14 Working in line mode. 14 Working in line mode. 14	Running an X-Window application	4
Security	The shell user	4
Accessing the shells – the choices	Security	4
Interoperability between the shells and MVS	Accessing the shells – the choices	
Interoperability between the Shells and MVS. 6 Parallels between the MVS environment and the shell environment. 7 Programming for everyday tasks. 8 Editing. 8 Job control. 9 Background jobs. 9 Programming. 9 Debugging. 9 Debugging. 9 Data management. 9 Chapter 2. OMVS, a 3270 terminal interface to the z/OS shell. 11 Differences from a UNIX or AIX environment. 11 Invoking the shell. 11 Chapter 2. OMVS, a 3270 terminal interface to the z/OS shell. 11 Differences from a UNIX or AIX environment. 11 Invoking the shell. 11 Chapting options on the OMVS command. 12 Understanding the shell screen. 12 Working in line mode. 14 Why isn't your output displayed on the screen? 14 Determining function key settings and the escape character. 15 The function key functions. 15 The escape character. 18 Entering a shell command. 18 Customizing the v	Ierminal emulators	
Parallels between the MVS environment and the shell environment. / Programming for everyday tasks. 8 Editing. 8 Job control. 9 Background jobs. 9 Programming. 9 Debugging. 9 Debugging. 9 Data management. 9 Chapter 2. OMVS, a 3270 terminal interface to the z/OS shell. 11 Differences from a UNIX or AIX environment. 11 Invoking the shell. 11 Chapter 3. OMVS on the OMVS command. 12 Understanding the shell screen. 12 Working in line mode. 14 Why isn't your output displayed on the screen? 14 Determining function key settings and the escape character. 15 The function key functions. 15 The escape character. 18 Entering a shell command. 18 Customizing the variant characters on your keyboard. 18 Entering a long shell command 18	Interoperability between the shells and MVS.	
Programming for everyday tasks. 8 Editing. 8 Job control. 9 Background jobs. 9 Programming. 9 Debugging. 9 Data management. 9 Chapter 2. OMVS, a 3270 terminal interface to the z/OS shell. 11 Differences from a UNIX or AIX environment. 11 Invoking the shell. 11 Changing options on the OMVS command. 12 Understanding the shell screen. 12 Working in line mode. 14 Why isn't your output displayed on the screen? 14 Determining function key settings and the escape character. 15 The function key functions. 15 The escape character. 18 Entering a shell command. 18 Customizing the variant characters on your keyboard. 18 Entering a long shell command. 18	Parallels between the MVS environment and the shell environment	
Background jobs. 9 Background jobs. 9 Programming. 9 Debugging. 9 Data management. 9 Chapter 2. OMVS, a 3270 terminal interface to the z/OS shell. 11 Differences from a UNIX or AIX environment. 11 Invoking the shell. 11 Changing options on the OMVS command. 12 Understanding the shell screen. 12 Working in line mode. 14 Why isn't your output displayed on the screen? 14 Determining function key settings and the escape character. 15 The function key functions. 15 The scape character. 18 Entering a shell command. 18 Customizing the variant characters on your keyboard. 18	Programming for everyday tasks	8 8 م
Job control. 9 Background jobs. 9 Programming. 9 Debugging. 9 Data management. 9 Chapter 2. OMVS, a 3270 terminal interface to the z/OS shell. 11 Differences from a UNIX or AIX environment. 11 Invoking the shell. 11 Changing options on the OMVS command. 12 Understanding the shell screen. 12 Working in line mode. 14 Why isn't your output displayed on the screen? 14 Determining function key settings and the escape character. 15 The function key functions. 15 The escape character. 18 Entering a shell command. 18 Customizing the variant characters on your keyboard. 18 Entering a long shell command 18	Ealling	۵۵
Background jobs	JOD CONTROL	
Programming. 9 Debugging. 9 Data management. 9 Chapter 2. OMVS, a 3270 terminal interface to the z/OS shell. 11 Differences from a UNIX or AIX environment. 11 Invoking the shell. 11 Changing options on the OMVS command. 12 Understanding the shell screen. 12 Working in line mode. 14 Why isn't your output displayed on the screen? 14 Determining function key settings and the escape character. 15 The function key functions. 15 The escape character. 18 Entering a shell command. 18 Customizing the variant characters on your keyboard. 18 Entering a long shell command. 18	Dackgrouniu jobs	
Debugging	Programming	
Data management	Debugging	
Chapter 2. OMVS, a 3270 terminal interface to the z/OS shell.11Differences from a UNIX or AIX environment.11Invoking the shell.11Changing options on the OMVS command.12Understanding the shell screen.12Working in line mode.14Why isn't your output displayed on the screen?14Determining function key settings and the escape character.15The function key functions.15The escape character.18Entering a shell command.18Entering a long shell command.18	Data Indiagement	9
Differences from a UNIX or AIX environment. 11 Invoking the shell. 11 Changing options on the OMVS command. 12 Understanding the shell screen. 12 Working in line mode. 14 Why isn't your output displayed on the screen? 14 Determining function key settings and the escape character. 15 The function key functions. 15 The escape character. 18 Entering a shell command. 18 Entering a long shell command 18	Chapter 2 $OMVS = 3270$ terminal interface to the z/OS shall	11
Invoking the shell. 11 Changing options on the OMVS command. 12 Understanding the shell screen. 12 Working in line mode. 14 Why isn't your output displayed on the screen? 14 Determining function key settings and the escape character. 15 The function key functions. 15 The escape character. 18 Entering a shell command. 18 Entering a long shell command. 18 Entering a long shell command. 18	Differences from a UNIX or AIX environment	11
Changing options on the OMVS command. 12 Understanding the shell screen. 12 Working in line mode. 14 Why isn't your output displayed on the screen? 14 Determining function key settings and the escape character. 15 The function key functions. 15 The escape character. 18 Entering a shell command. 18 Entering a long shell command. 18 Entering a long shell command. 18	Invoking the shell	11
Understanding the shell screen	Changing ontions on the OMVS command	12
Working in line mode	Understanding the shell screen	
Why isn't your output displayed on the screen? 14 Determining function key settings and the escape character 15 The function key functions 15 The escape character 18 Entering a shell command 18 Customizing the variant characters on your keyboard 18 Entering a long shell command 18	Working in line mode	۲۲ ۱۸
Determining function key settings and the escape character	Why isn't your output displayed on the screen?	14 14
The function key functions. 15 The escape character. 18 Entering a shell command. 18 Customizing the variant characters on your keyboard. 18 Entering a long shell command 18 Image: State of the sta	Determining function key settings and the escape character	
The escape character	The function key functions	
Entering a shell command	The escape character	
Customizing the variant characters on your keyboard	Entering a shell command	
Entering a long shell command 18	Customizing the variant characters on your keyboard	
	Entering a long shell command	

	10
Entering a shell command from ISU/E	
Interrupting a shell command	
Typing escape sequences in the shell	19
Suppressing the newline character	
Keyboard remapping	20
Determining your session status	
Scrolling through output	
Using function keys or subcommands	
Using cursor scrolling	
Running a subcommand	22
Switching to subcommand mode	22
Liging multiple sessions	
Starting cossions	
Suitching between cossions	
Switching between sessions	
Customizing the OMVS internace	
An example of customizing the UMVS command	
The alarm setting (ALARM NOALARM)	
Autoscrolling (AUTOSCROLL NOAUTOSCROLL)	
The character conversion table (CONVERT)	
Double-byte character set support (DBCS NODBCS)	24
Debugging for the OMVS command (DEBUG)	
Giving an application control of the command line (ECHO NOECHO)	24
Ending 3270 pass-through mode (ENDPASSTHROUGH)	
The escape character (ESCAPE)	
Controlling the size of the output scroll buffer (LINES)	
Function key settings (PEn).	
Displaying the function key settings (PESHOW NOPESHOW)	25
Specifying Language Environment runtime ontions (RUNOPTS)	20
Multiple sessions (SESSIONS)	
The shared TSO/E address space (SUAPEAS NOSUAPEAS)	
Controlling data recorded in the debug data set (MDADDEBLIC)	
Controlling data recorded in the debug data set (WRAPDEDOG)	
Performing ISO/E work or ISPF work after invoking the shell	
Entering a ISO/E command from the Z/OS shell	
Switching to ISO/E command mode	
ttp or telnet from TSO	
Exiting the shell	
Getting rid of a hung application	
Using a double-byte character set (DBCS)	
Single-byte restrictions	
Chapter 3. The asynchronous terminal interface to the shells	
ASCII-EBCDIC translation	
Using rlogin to access the shell	
Using telnet to access the shell	
Using Communications Server login to access the shell	
The shell session	32
Entering a shell command	32
Interrunting a shell command	32
Light multiple socions	20 21
Using a double byte obstactor set (DBCC)	
Using a double-byte character set (DBCS)	
Standard shell escape characters	
Chapter 4. Customining the = 100 shall	35
Chapter 4. Customizing the Z/US shell	
Quoting variable values	
Changing variable values dynamically	
Understanding shell variables	
Customizing your shell environment: The ENV variable	

Adding your working directory to the search path	
Checking the search path used for a command	
Customizing the FPATH search path: The FPATH variable	
Customizing the DLL search path: The LIBPATH variable	
Improving the performance of shell scripts	4
Changing the locale in the shell	А
Advantages of a locale compatible with the MVS code page	Δ.
Advantages of a locale generated with code page IBM-10/7	
Changing the locale setting in your profile	
The IC SVNTAX environment variable	.+
The LOCPATH environment variable	
Customizing the language of your messages	
Sotting your local time zone	44 ۸
Duilding a STEDLID anvironment. The STEDLID anvironment veriable	
Duilding a STEPLID environment. The STEPLID environment variable	
Restrictions on STEPLIB data sets	
Setting options for a shell session	
Exporting variables	
Controlling redirection	
Preventing wildcard character expansion	
Displaying input from a file	
Running a command in the current environment	
Displaying current option settings	41
Chapter 5. Customizing the tcsh shell	
Understanding the startup files	
Quoting variable values	
Changing variable values dynamically	
Understanding shell variables	
Customizing your shell environment: The .tcshrc file	
Customizing the search path for commands: The PATH variable	
Adding your working directory to the search path.	53
Checking the search path used for a command	5
Customizing the DIT search nath: The LIBPATH variable	5/
Changing the locale in the shell	
Advantages of a locale compatible with the MVS code page	-0
Advantages of a locale compatible with the MVS code page	
Changing the locale setting in your profile	
The LC SYNTAX environment variable	
The LOCDATH environment variable	
The LOCPATH environment variable	
Customizing the language of your messages	
Setting your local time zone	
Building a STEPLIB environment: The STEPLIB environment variable	
Restrictions on STEPLIB data sets	
Setting variables for a shell session	59
Displaying current option settings	59
Controlling redirection	
Preventing wildcard character expansion	
Displaying input from a file	
Displaying deletion verification	
Files accessed at termination	
hapter 6. Working with z/OS shell commands	
Specifying shell command ontions	
opeonying onen commune options	
Specifying options with accompanying arguments	
Specifying options with accompanying arguments	6
Specifying options with accompanying arguments Help for shell command usage	۵۵ ۲۷

	Redirecting input from a file.	.64
	Redirecting error output to a file	64
	Closing a file	64
	Dumping nontext files to standard output	65
	Setting up an alias for a command	65
	Dofining an alias	.05
	Dedefining an alias for a cossion	.05
	Cetting up on plice for a particular version of a command	. 00
	Setting up an allas for a particular version of a command	. 66
	Using alias tracking	.67
	Iurning off an allas	.68
	Combining commands	.68
	Using a semicolon (;)	.68
	Using && and	. 68
	Using a pipe	.69
	Using substitution in commands	69
	Using the find command in command substitution constructs	.70
	Characters that have special meaning to the shell	. 70
	Characters used with commands	.71
	Characters used in file names	. 71
	Redirecting input and output	.72
	Using a special character without its special meaning	. 73
	The backslash	73
	A pair of single quotation marks (' ')	.73
	A pair of double quotation marks (" ")	.74
	Using a wildcard character to specify file names	.74
	The * character	.74
	The ? character	.74
	The square brackets	75
	Retrieving previously entered commands	. 75
	Retrieving commands from the history file	.76
	Editing commands from the history file	. 76
	Using the retrieve function keys	.77
	Command-line editing	. 77
	Using record-keeping commands	.79
	Finding elements in a file and presenting them in a specific format	. 79
	Timing programs	.80
	Using the passwd command	. 80
	Switching to superuser or another ID	.80
	Using the whoami command	.81
	Running a TSO/E command	81
	Using the tso command	.81
	Using the tsocmd command	.82
	Using the man command to get online help	.82
	Shell messages	.83
Ch	apter 7. Working with tcsh shell commands	.85
	Specifying shell command options	.85
	Specifying options with accompanying arguments	.86
	Help for shell command usage	.86
	Understanding standard input, standard output, and standard error	. 86
	Redirecting command output to a file	87
	Redirecting input from a file	.88
	Redirecting error output to a file	.88
	Dumping nontext files to standard output	.88
	Setting up an alias for a command	.88
	Defining an alias	.89
	Redefining an alias for a session	.89
	Setting up an alias for a particular version of a command	. 90
		2

Turning off an alias	90
Combining commands	
Using a semicolon (;)	
Using && and	
Using a pipe	92
Using substitution in commands	
Using the find command in command substitution constructs	93
Characters that have special meaning to the shell	
Characters used with commands	93
Characters used in file names	
Redirecting input and output	95
Using a special character without its special meaning	
The backslash	
A pair of single quotation marks (' ')	96
A pair of double quotation marks (" ")	96
Using a wildcard character to specify file names	96
The * character	96
The ? character	
The square brackets	
Retrieving previously entered commands	
Retrieving commands from the history file	
Editing commands from the history file	
Using the retrieve function keys	
Command-line editing	
Using file name completion	
Using record-keeping commands	101
Finding elements in a file and presenting them in a specific format	
Timing programs	
Using the passwd command	
Switching to superuser or another ID	103
Using the whoami command	
Running a TSO/E command	
Using the tso command	104
Using the tsocmd command	105
Online help	
Using the man command	105
Shell messages	105
Chapter 8. Writing z/OS shell scripts	
Running a shell script	
Using the magic number	
Using TSO/E commands in shell scripts	
Using variables	
Creating a variable	
Calculating with variables	
Exporting variables	
Associating attributes with variables	
Displaying currently defined variables	
Using positional parameters – the \$N construct	
Using quotation marks to enclose a construct in a shell script	
Using parameter and variable expansion	
Using special parameters in commands and shell scripts	
Using control structures	
Using the test command to test conditions	
The unbild loop	
The for loop	121
Combining control structures	121
compining control structures	

Using functions	122
Autoloading functions	123
Chapter 9. Writing tcsh shell scripts	125
Running a shell script	125
Using the magic number	126
Using TSO/E commands in shell scripts	126
Using variables	126
Creating a shell variable	126
Calculating with variables	127
Setting environment variables	128
Using positional parameters — the \$N construct	129
Using quotes to enclose a construct in a shell script	130
Using parameter and variable expansion	131
Using special parameters in commands and shell scripts	131
Using control structures	132
The if conditional	132
The while loop	133
The foreach loop	134
Combining control structures.	
Chapter 10. Using job control in the shells.	
Running several jobs at once (foreground and background).	
Starting a job in the background with an ampersand (&)	136
Moving a job to the background	136
Moving a job to the foreground	136 1
Setting up ioh tracing	136
Checking the status of jobs	137
Using the ichs command	<u>1</u> 37
Using the ps command	137
Canceling a job	<u>1</u> 37 138
Canceling a foreground ich	138
Canceling a hackground job	138
Stopping and resuming a job	130
Stopping and resuming a job	130
Stopping a background job	139 130
Recuming a stopped job	120
Delaving a command	139 130
Eviting the shell with background jobs running	120
Changing the default in the z/OS shall	1/0
Comparison of shall background jobs and MVS batch jobs	140 1
	140
Chapter 11 Using 2/OS UNIX from batch TSO/E and ISPE	1/12
TCL support for z/OS UNIX	1/12
The DATH knowerd	1/2
The DENITYDE knowed	1 / 1
Light the deferme in an application	1 / 1 /
Specifying a depage in the ICL	144
Specifying a duriante in the JCL	145 145
Using the submit command	145 1 4 F
	145
Allases for BPXBATCH.	146
Defining standard input, output, and error streams for BPXBAICH	146
Passing environment variables to BPABAICH	147
Passing parameter data to BPXBAICH	149
INVOKING BYXBAICH IN A DATCH JOD	151
Invoking BPXBAICH from the ISU/E environment	154
USING ISU/E REXX TOR Z/US UNIX processing	156
Using the ISPF shell	157

Invoking the ISPF shell	
Working in the ISPF shell	
Using the online help facility	
Chapter 12. Performance: Running executable files	
Improving shell script performance	
Chapter 13. Communicating with other users	163
Using mailx to send and receive mail	
Steps for sending mail to another user	163
Sending mail to a distribution list	
Sending a message to an MVS operator	
Receiving mail from other users	
Replying to mail	
Saving and deleting mail	
Ending the mailx program	
Using write to send a message or a file	
Sending a message: An example	
Ending a message	166
Sending a file	
Using talk for an online conversation	
Beginning a conversation: An example	167
Viewing the conversation	
Using wall to broadcast messages	
Controlling messages and online conversations	168
Using the UUCP network	168
Transferring a file to a remote site	
Transferring multiple files to a remote site	
I ransferring a file to the local public directory	
Notification of transfer	
Permissions.	
I ransferring a file from a remote site	
Unecking a file s transfer status.	⊥/⊥ 1⊓1
Working with your files in the public directory	172
Running a command on a remote site	172
Using TSO/E to serio of receive mail	172 (L
Sending a message to a distribution list	172
Sending a message to an MVC operator	1/3 172
Becoiving mail from other users	1/3 172
Receiving mail from other systems	1/3 172
Receiving messages nom other systems	1/3
Part 2. The z/OS UNIX file system	175
· · ·	_
Chapter 14. An introduction to the z/OS UNIX file system	
The root file system and mountable file systems	
Directories	178
Files	
Files not in the file system	179
Comparison between MVS data sets and the z/OS UNIX file system	
Sharing files between LPARs	
Executable modules in the file system	
Path and path name	
Requirement for an absolute path name	
Resolving a symbolic link in a path name	
Command differences with symbolic links	
Using commands to work with directories and files	

Entering a TSO/E command	
Using a relative path name on TSO/E commands	185
Finding the data set that contains a file	185
Using the ISPF shell to work with directories and files	186
Using the Network File System feature	
External links	
Security for the file system	
The file system and power failures	186
Chapter 15. Converting files between code pages	
Enhanced ASCII	
File tagging in Enhanced ASCII	
Unicode Services	
File tagging in Unicode Services	190
Automatic code set conversion	
Porting considerations	
· · · · · · · · · · · · · · · · · · ·	
Chapter 16. Working with directories.	
The working directory	191
Displaying the name of your working directory	191
Changing directories	192
Using notations for relative nath names	192
Creating a directory	193
Removing a directory	195
Listing directory contents	195
Comparing directory contents	196
Finding a directory or file	
Chapter 17 Working with files	100
Lising an editor to create a file	100
Naming files	100
Processing in uppercase and lowercase	200
Deleting a file	200
Deleting files over a certain age	201
Identifying a file by its incde number	201
Croating links	202
Creating times	202
Creating a nard link	202
Creating a symbolic link.	203
	204
Deterning uniks	205
Comparing files	205
Companing file contents	
Soluting file contents	200
Counting lines, words, and bytes in a file	207
Counting tiles, words, and bytes in a file	
Searching mes by using pattern matching	
	209
Regular expressions	
Browsing files.	
Browsing files without formatting	
Browsing files with formatting	
Simultaneous access to a file	
Backing up and restoring files: options	
Backing up and restoring files from the shell	212
Backing up a complete directory into an MVS data set	
Restoring a complete directory from an MVS data set	
Viewing the contents of an archive	
Converting between code pages	

Appending to an existing archive	215
Backing up selected files by date	216
Listing process IDs of processes with open files	216
	015
Chapter 18. Handling security for your files	
Changing a survisions set by the system	
Changing permissions for files and directories	
Using a symbolic mode to specify permissions	
Using octal numbers to specify permissions	
Using the sticky bit on a directory to control file access	
Auditing file access	222
Displaying file and directory permissions	223
Setting the file mode creation mask	
Changing the owner ID or group ID associated with a file	224
Temporarily changing the user ID or group ID during execution	225
Displaying extended attributes	
Using access control lists (ACLs) to control access to files and directories	
Setting up ACL support	
Chapter 19. Editing files	
Using ISPF to edit a Z/US UNIX file	
Using the vi screen editor	
Basic principles	
A simple vi session	
Adding text	
Moving the cursor up and down the screen	
Moving up and down through a file	
Moving the cursor on the line	
Moving to sentences and paragraphs	
Deleting text	233
Changing text	
Undoing a command	233
Saving a file	234
Searching for strings	234
Moving text	236
Copying text	
Other vi features	
Message: vi/ex edited file recovered	237
Using the ed editor	239
Creating and saving a text file	239
Editing an existing file	240
Identifying line numbers and changing your position in the buffer	
Appending one file to another	
Displaying the current line in the edit buffer	
Changing a character string	241
Inserting text at the beginning or end of a line	
Deleting lines of text	
Changing lines of text	
Inserting lines of text	243
Copving lines of text	
Moving lines of text	
Undoing a change	
Entering a shell command while using ed	
Ending an ed edit session	244
Default permissions	
Using sed to edit a z/OS UNIX file	
Chapter 20. Printing files	

Formatting files for online browsing or printing Printing requests in shell scripts	.247
Printing with the In command	2/12
Printing with TSO/E commands	2/18
Checking the status of print jobs	2/10
	. 247
Chapter 21. Copying data between the z/OS UNIX file system and MVS data sets	. 251
Copying data using z/OS shell commands	. 251
Copying data using TSO/E commands	251
Copying a sequential data set or PDS member into a z/OS UNIX file	252
Using cp to copy a sequential data or PDS member into a z/OS UNIX file	. 253
Using OPUT and OCOPY to copy a PDS member, a PDSE member, or a sequential data set	.253
Copying a PDS or PDSE to a z/OS UNIX directory	. 256
Using cp to copy a PDS to a z/OS UNIX directory	. 256
Using OPUTX to copy a sequential data set or members of a PDS or PDSE	. 256
Copying an MVS VSAM data set to a z/OS UNIX file	.257
Copying a z/OS UNIX file into a sequential data set or PDS member	257
Using cp to copy a z/OS UNIX file into a sequential data set or PDS member	. 258
Using OGET and OCOPY to copy a file into a sequential data set or a PDS member	258
OGET	. 258
OCOPY	.259
Copying z/OS UNIX files into a PDS or PDSE	.261
Using cp to copy z/OS UNIX files into a PDS or PDSE	261
Using OGETX to copy files into a PDS or PDSE	. 261
Copying files within the z/OS UNIX file system	.262
Copying an MVS data set into another MVS data set	. 263
Example: Using ALLOCATE and OCOPY	. 264
Example: Using JCL and OCOPY	. 264
Copying executable modules between MVS data sets and the z/OS UNIX file system	.264
Using cp to copy executables between MVS and z/OS UNIX	.264
Using TSO/E commands and JCL to copy executables	. 264
Copying data: Code page conversion	266
Single-byte data	266
Double-byte data	.267
Chapter 22 Transferring files between systems	269
File transfer directly to or from z/OS UNIX	269
Transferring files using File Transfer Protocol (FTP)	269
Transferring files using the Network File System feature	269
Transferring files using the SEND and RECEIVE programs	269
Transferring files by using the File Transfer, Access, and Management Function.	270
File transfer using MVS data sets	.270
Transferring files into the z/OS UNIX file system	. 270
Transferring files to the workstation	.270
Transporting an archive file on tape or diskette	.270
Putting an archive file into the file system	.271
Sending an archive file to others	.272
Appendix A. Advanced vi topics	275
	.275
Setting tab stops	. 275
Using appreviations	.275
Utner ealting options	. 275
Setting up an editing options command file	.276
Editing several files	.276
	. 277
Eating program source code	

Controlling indention	
Searching for opening and closing brackets	
Making substitutions	
ppendix B. Using awk	
Data files	
Records	
Fields	
The shape of a program	
Simple patterns	
Using blanks and horizontal tabs	
Applying more than one instruction	
Assigning values to variables	
String values	
Numeric values	
Using the print action for output	
Running awk programs	
The awk command line	
Program files	
Sources of data	
Operators	
Comparison operators	
Arithmetic operators	
Compound assignments	
Increment and decrement operators	
Matching operators	
Multiple-condition operators	
Regular expressions	
Pattern ranges	
Using special patterns	
Built-in variables	
Built-in numeric variables	
Built-in string variables	
Statements and loops	
The if statement	
The while loop	
The for loop.	
The next statement	
The exit statement	
Functions	
Arithmetic functions	
String manipulation functions	
User-defined functions	
Passing an array to a function	
The Getline function	
Running system commands	
Controlling awk output	
Formatting the output	30
Placeholders	
Escape sequences	20

Appendix C. Code page conversion when the shell and MVS have different

locales	
Customizing the variant characters on your keyboard	
Using the CONVERT option on the OMVS command	
When do you need to convert between code pages?	
Methods for converting data	306

Acknowledgments	
Trademarks	
Programming interfaces	324
Minimum supported hardware	324
Policy for unsupported hardware	323
IBM Online Privacy Statement	323
Terms and conditions for product documentation	322
Notices	321
Dotted decimal syntax diagrams	317
Keyboard navigation of the user interface	
Consult assistive technologies.	
Accessibility features	317
Annendix F. Accessibility	317
Appendix E. Locale objects, source files, and charmaps	315
BPXFX437, BPXFX450, BPXFX471, BPXFX473, BPXFX477, BPXFX478, BPXFX480, BPXFX484, BPXFX485, BPXFX497 conversion tables	
BPXFX111 and BPXFX211 conversion tables	312
BPXFX100 conversion table	
Escape sequences unique to a conversion table	
Escape sequences for control characters	
Appendix D. Escape sequences for a 3270 keyboard	309 309
	200
The POSIX portable character set	
The PUSIX portable file name character set	

List of Figures

1. How the shells fit into z/OS	3
2. The OMVS interface to the shell	6
3. The asynchronous terminal interface to the shell	6
4. z/OS UNIX System Services provides the user interfaces of both MVS and UNIX	7
5. Working interactively in the MVS and shell environments	8
6. Switching temporarily to TSO/E command mode or subcommand mode	
7. The z/OS shell's display screen when the shell is first invoked	13
8. The z/OS shell's display screen after input has been entered	14
9. A sample .tcsh file	
10. The OSHELL REXX exec	156
11. ISPF shell: The main panel	158
12. End user's logical view of the file system	177
13. Organization of the file system	177
14. Comparison of MVS data sets and the z/OS UNIX file system	180
15. Creating a new directory	194
16. Hard link: a new name for an existing file	203
17. Symbolic link: a new file	203
18. External link: A new file	204
19. A sample file: comics.lst	207
20. Copying data between z/OS UNIX and MVS	251

List of Tables

1. Function key settings available in the z/OS shell	15
2. Three ways to set the STEPLIB environment variable (z/OS shell)	45
3. Three ways to set the STEPLIB environment variable (tcsh shell)	58
4. Using the test command to examine the nature of a file	118
5. Using the test command to compare the age of two files	118
6. Using the test command to compare the values of two numbers	118
7. Using the test command to compare two strings	118
8. Using the test command to test whether strings are empty	118
9. Comparison of running a background job from the shell and from MVS	140
10. Requirements for absolute path names	182
11. Three-digit permissions specified in octal	221
12. vi editor: Positioning the cursor	229
13. Portable characters: Escape sequences	309
14. Control characters: Escape sequences	310
15. Translation of selected escaped characters (BPXFX100)	311
16. Translation of selected escaped characters (BPXFX111 and BPXFX211)	312
17. Translation of selected escaped characters (BPXFX437, BPXFX450, BPXFX471, BPXFX473, BPXFX477, BPXFX478, BPXFX480, BPXFX484, BPXFX485, and BPXFX497)	312

About this document

This document provides an introduction to the two shells that are available on z/OS UNIX System Services (z/OS UNIX) — the z/OS shell and the tcsh shell.

This document provides the information you need to use the z/OS Shells and Utilities on an IBM[®] z/OS[®] system. The Shells and Utilities and TSO/E (Time Sharing Option Extensions) provide commands for using z/OS UNIX.

This document helps you 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, it also describes other z/OS UNIX support services.

Who should use z/OS UNIX System Services User's Guide?

This document is for application programmers, systems programmers, and users working on a z/OS system and using z/OS UNIX services or the z/OS shells.

This document assumes that readers are familiar with the z/OS system and with the information for z/OS and its accompanying products.

What is in z/OS UNIX System Services User's Guide?

This document describes how to use the shells, the file system, and communication services. Using the document, you can:

- 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 Clanguage program.
- Run shell scripts and C language programs interactively (in the foreground), in the background, or in batch.
- Switch easily between the shells and TSO/E.
- Move MVS[™] data sets into the file system, or move files from the file system into MVS data sets.
- Enter shell commands or TSO/E commands from the shell command line.
- Create or edit a file in the file system.
- Manage your file system.

For a discussion of the z/OS UNIX shell commands, utilities, TSO/E commands, and file formats, see <u>z/OS</u> UNIX System Services Command Reference.

Tasks that can be performed in more than one environment

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

z/OS information

This information explains how z/OS references information in other documents and on the web.

When possible, this information uses cross document links that go directly to the topic in reference using shortened versions of the document title. For complete titles and order numbers of the documents for all products that are part of z/OS, see z/OS Information Roadmap.

To find the complete z/OS library, go to IBM Knowledge Center (www.ibm.com/support/knowledgecenter/SSLTBW/welcome).

Discussion list

Customers and IBM participants also discuss z/OS UNIX on the mvs-oe discussion list. This list is not operated or sponsored by IBM.

To subscribe to the mvs-oe discussion, send a note to:

listserv@vm.marist.edu

Include the following line in the body of the note, substituting your given name and family name as indicated:

subscribe mvs-oe given_name family_name

After you have been subscribed, you will receive further instructions on how to use the mailing list.

How to send your comments to IBM

We invite you to submit comments about the z/OS product documentation. Your valuable feedback helps to ensure accurate and high-quality information.

Important: If your comment regards a technical question or problem, see instead <u>"If you have a technical</u> problem" on page xxi.

Submit your feedback by using the appropriate method for your type of comment or question:

Feedback on z/OS function

If your comment or question is about z/OS itself, submit a request through the <u>IBM RFE Community</u> (www.ibm.com/developerworks/rfe/).

Feedback on IBM Knowledge Center function

If your comment or question is about the IBM Knowledge Center functionality, for example search capabilities or how to arrange the browser view, send a detailed email to IBM Knowledge Center Support at ibmkc@us.ibm.com.

Feedback on the z/OS product documentation and content

If your comment is about the information that is provided in the z/OS product documentation library, send a detailed email to <u>mhvrcfs@us.ibm.com</u>. We welcome any feedback that you have, including comments on the clarity, accuracy, or completeness of the information.

To help us better process your submission, include the following information:

- · Your name, company/university/institution name, and email address
- The following deliverable title and order number: z/OS UNIX System Services User's Guide, SA23-2279-30
- · The section title of the specific information to which your comment relates
- The text of your comment.

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

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

If you have a technical problem

If you have a technical problem or question, do not use the feedback methods that are provided for sending documentation comments. Instead, take one or more of the following actions:

- Go to the IBM Support Portal (support.ibm.com).
- Contact your IBM service representative.
- Call IBM technical support.

Summary of changes

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

Summary of changes for z/OS UNIX for Version 2 Release 3

Changed

- These figures were updated to include the new /global directory:
 - Figure 12 on page 177
 - Figure 13 on page 177

Deleted

• Because the z/OS UNIX System Services website is no longer available, information about it was deleted. Information about Tools and Toys was also deleted.

Summary of changes for z/OS UNIX for Version 2 Release 2 (V2R2)

Changed

• Additional information was added to "Naming files" on page 199.

z/OS Version 2 Release 1 summary of changes

See the Version 2 Release 1 (V2R1) versions of the following publications for all enhancements related to z/OS V2R1:

- z/OS Migration
- z/OS Planning for Installation
- z/OS Summary of Message and Interface Changes
- z/OS Introduction and Release Guide

Part 1. The z/OS shells

These topics provide an overview of the information relating to the z/OS shells.

Chapter 1. An introduction to the z/OS shells

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

- The z/OS shell.
- The tcsh shell.

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

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

Figure 1 on page 3 shows how these shells fit into z/OS.



Figure 1: How the shells fit into z/OS

About shells

A shell is a command interpreter that you use to:

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

Shell commands and utilities

Both the z/OS shell and the tcsh shell provide commands and utilities that give the user an efficient way to request a range of services. In this topic, the term *command* is used to include both a *command* (a directive to a shell to perform a specific task) and a *utility* (the name of a program callable by name from a shell).

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

ls -a myfiles

1s is the command name, -a is the option, and *myfiles* is the argument.

This information describes various commands you can use to perform certain tasks; most of these are shell commands, and some are TSO/E commands. This discussion highlights only certain functions of the command. For complete information about each command and all its options, see <u>z/OS UNIX System</u> Services Command Reference.

The locale in the shells

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

The shells and utilities support a variety of locales. See <u>"Changing the locale in the shell" on page 40</u> for information about changing the locale in the shells.

Daemon support

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

• For information about each daemon that z/OS UNIX System Services provides, see <u>z/OS UNIX System</u> Services Command Reference.

Running an X-Window application

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

The shell user

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

Security

This information assumes that your system includes the RACF[®] security product. Instead of RACF, your system could have an equivalent security product.

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

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

Accessing the shells – the choices

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

LISTUSER USERNAME OMVS

The RACF settings for that user are displayed:

```
UID= 000000012
HOME= /shut/home/billyjc
PROGRAM= /bin/sh
CPUTIMEMAX= NONE
ASSIZEMAX= NONE
FILEPROCMAX= NONE
PROCUSERMAX= NONE
THREADSMAX= NONE
MMAPAREAMAX= NONE
READY
```

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

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

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

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

Terminal emulators

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

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

When you select a terminal emulator, consider these key points:

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

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

line mode

Your input is processed after you press <Enter>.

raw mode

Each character is processed as you type it.

graphical mode

A graphical user interface for X-Window applications

Terminal Interface



Figure 2: The OMVS interface to the shell

Terminal Interface





Interoperability between the shells and MVS



Figure 4: z/OS UNIX System Services provides the user interfaces of both MVS and UNIX

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

- You can move data between MVS data sets and the z/OS UNIX file system. You can copy or move MVS data sets into z/OS UNIX files; likewise, you can copy or move z/OS UNIX files into MVS data sets.
- To work with z/OS UNIX files, you can use TSO/E commands or shell commands. If you have access to ISPF, you can use the panel interface of the ISPF shell. For example, you can create a directory with the TSO/E MKDIR command, or the shell mkdir command, or the ISPF shell.
- You can issue TSO/E commands from the shell command line, from a shell script, or from a program. See <u>"Using commands to work with directories and files" on page 184</u> for a list of TSO/E commands you can use to work with the file system.
- You can write job control language (JCL) that includes shell commands.
- To edit z/OS UNIX files, you can use the ISPF/PDF full-screen editor or one of the editors available in the shell.
- REXX programs can access kernel callable services by using z/OS UNIX extensions to the REstructured eXtended eXecutor (REXX) language. You can run REXX programs using these extensions from TSO/E, batch, the shell, or a C program.
- Use the OSHELL REXX exec to run a non-interactive shell command or shell script from the TSO/E READY prompt and display the output to your terminal. This exec is shipped with z/OS UNIX System Services.

Parallels between the MVS environment and the shell environment

Figure 5 on page 8 indicates how basic programming tasks are performed in the MVS environment and in the shell environment.

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

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



Figure 5: Working interactively in the MVS and shell environments

Programming for everyday tasks

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

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

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

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

Editing

In MVS, you can edit z/OS UNIX files by using the TSO/E OEDIT command to invoke ISPF File Edit or by selecting File Edit on the ISPF menu, if it is installed.

In a shell, you can use the ed and sed editors for editing z/OS UNIX files. You can use the oedit shell command to invoke ISPF File Edit. If you use rlogin or telnet to login to the shell, you can also use the vieditor.

Job control

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

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

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

Background jobs

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

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

Programming

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

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

Debugging

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

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

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

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

The dbx debugger also provides core dump analysis when run in dump processing mode.

Data management

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

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

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

Chapter 2. OMVS, a 3270 terminal interface to the z/OS shell

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

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

You issue the OMVS command from TSO/E:

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

For information about using an asynchronous terminal interface to the shell, see <u>Chapter 3</u>, "The asynchronous terminal interface to the shells," on page 31.

Differences from a UNIX or AIX environment

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

OMVS interface	For more information
The 3270 interface operates in line mode (also known as canonical mode). You type data on a command line and no data is transmitted until you press the <enter> key.</enter>	"Working in line mode" on page 14
The 3270 interface has function keys for various tasks such as scrolling through output, running TSO/E commands, and so on.	"Determining function key settings and the escape character" on page 15
The OMVS interface does not have a control key. Instead of using a <ctrl> key to type control sequences (for example, <ctrl-d>), you use the Control function key or a multicharacter escape-key sequence.</ctrl-d></ctrl>	<u>"Typing escape sequences in the shell"</u> on page 19
With the OMVS interface, you can edit z/OS UNIX files using the ISPF editor or the ed editor. Because this interface runs in line mode, you cannot use the vi editor.	Chapter 19, "Editing files," on page 227
Delayed display of output: If a command you are running does not produce output for more than a few seconds, you will need to repeatedly press the Refresh key to display the output as it is produced.	"Why isn't your output displayed on the screen?" on page 14

Invoking the shell

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



Figure 6: Switching temporarily to TSO/E command mode or subcommand mode

To invoke the shell:

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

The systems programmer might have set up your TSO/E user's logon to invoke the shell automatically. In that case, you do not need to perform step 2.

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

Changing options on the OMVS command

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

You can create a customized version of the OMVS command for your own use, by writing a simple REXX program or CLIST that specifies certain keywords on the command. For information about how to do this, see <u>"Customizing the OMVS interface" on page 23</u> and <u>"An example of customizing the OMVS command"</u> on page 23.

Understanding the shell screen

When you start the shell, you see the panel in Figure 7 on page 13.

```
IBM
Licensed Material - Property of IBM
5647-A01 (C) Copyright IBM Corp. 1993, 2013
(C) Copyright Mortice Kern Systems, Inc., 1985, 1996.
(C) Copyright Software Development Group, University of Waterloo, 1989.
U.S. Government users - Restricted Rights
Use, duplication, or disclosure restricted by
GSA-ADP schedule contract with IBM Corp.
IBM is a registered trademark of the IBM Corp.
$
S
ESC=$ 1=Help 2=SubCmd 3=HlpRetrn 4=Top 5=Bottom 6=TS0
7=BackScr 8=Scroll 9=NextSess 10=Refresh 11=FwdRetr 12=Retrieve
```

Figure 7: The z/OS shell's display screen when the shell is first invoked

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

You can define a different prompt in your HOME/.profile file, if you want to. (See <u>Chapter 4</u>, <u>"Customizing the z/OS shell," on page 35</u> for more information about your <math>HOME/.profile file.)

You see:

- The command line (===>), used for input.
- The current function key settings and the current escape character assignments. You can turn off the function key display by typing the NOPF subcommand and turn on the display by typing the PF subcommand; alternatively, you can customize a function key to control the display of the function key settings. See "Customizing the OMVS interface" on page 23 for details on customizing function keys.

Note: The figures in this topic show the default function key settings.

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

Figure 8 on page 14 shows how a screen would look after some input had been entered.

GTWXT-XT-X 2 SMITHA 0 0 0 Nov 19 15:16 doc -TW-TWXTWX 2 SMITHA 0 250 Nov 17 23:07 etc -TW-T-T 2 SMITHA 0 17 Nov 17 23:07 fora -TW-T-T 5 SMITHA 0 1605 Dec 3 16:38 port -TW-T-T 5 SMITHA 0 472 Nov 17 23:15 script dTWXT-XT-X 2 SMITHA 0 0 Nov 17 23:07 src dTWXT-XT-X 2 SMITHA 0 0 Dec 3 20:37 projecta * * * * * * ESC= 1=Help 2=SubCmd 3=HlpRetrn 4=Top 5=Bottom 6=TS0 7=BackScr 8=Scroll 9=NextSess 10=Refresh 11=FwdRetr 12=Retrieve	\$ ls -1 total 7	7	•	ONTTUA	~		0	-	04.05				
dIWXT-XT-X 2 SMITHA 0 0 0 NoV 19 15:16 doc -IW-IWXIWX 2 SMITHA 0 250 Nov 17 23:07 etc -IW-I-IT- 2 SMITHA 0 17 Nov 17 23:07 fora -IW-I-IT- 5 SMITHA 0 1605 Dec 3 16:38 port -IW-I-IT- 2 SMITHA 0 472 Nov 17 23:15 script dIWXI-XI-X 2 SMITHA 0 0 Nov 17 23:07 src dIWXI-XI-X 2 SMITHA 0 0 Nov 17 23:07 src dIWXI-XI-X 2 SMITHA 0 0 Dec 3 20:37 projecta \$	arwxr-x	(r-x	2	SMITHA	0		⊎ Dec	3	04:25	bin			
-rw-rwxrwx 2 SMITHA 0 250 Nov 17 23:07 etc -rw-r-r 2 SMITHA 0 17 Nov 17 23:07 fora -rw-r-r 5 SMITHA 0 1605 Dec 3 16:38 port -rw-r-r 2 SMITHA 0 472 Nov 17 23:15 script drwxr-xr-x 2 SMITHA 0 0 Nov 17 23:07 src drwxr-xr-x 15 SMITHA 0 0 Dec 3 20:37 projecta \$ ===> INPUT ESC=¢ 1=Help 2=SubCmd 3=HlpRetrn 4=Top 5=Bottom 6=TS0 7=BackScr 8=Scroll 9=NextSess 10=Refresh 11=FwdRetr 12=Retrieve	arwxr-x	(r-x	2	SMITHA	0			19	15:16	doc			
-rw-rr- 2 SMITHA 0 17 Nov 17 23:07 fora -rw-rr- 5 SMITHA 0 1605 Dec 3 16:38 port -rw-rr- 2 SMITHA 0 472 Nov 17 23:15 script drwxr-xr-x 2 SMITHA 0 0 0 Nov 17 23:07 src drwxr-xr-x 15 SMITHA 0 0 0 Dec 3 20:37 projecta \$ ===> INPUT ESC=¢ 1=Help 2=SubCmd 3=HlpRetrn 4=Top 5=Bottom 6=TS0 7=BackScr 8=Scroll 9=NextSess 10=Refresh 11=FwdRetr 12=Retrieve	-rw-rwx	rwx	2	SMITHA	0	25	0 Nov	17	23:07	etc			
-rw-rr- 5 SMITHA 0 1605 Dec 3 16:38 port -rw-rr- 2 SMITHA 0 472 Nov 17 23:15 script drwxr-xr-x 2 SMITHA 0 0 Nov 17 23:07 src drwxr-xr-x 15 SMITHA 0 0 Dec 3 20:37 projecta \$ ===> INPUT ESC=¢ 1=Help 2=SubCmd 3=HlpRetrn 4=Top 5=Bottom 6=TS0 7=BackScr 8=Scroll 9=NextSess 10=Refresh 11=FwdRetr 12=Retrieve	-rw-r	·r	2	SMITHA	0	1	7 Nov	17	23:07	iora			
-rw-rr- 2 SMITHA 0 472 Nov 17 23:15 script drwxr-xr-x 2 SMITHA 0 0 Nov 17 23:07 src drwxr-xr-x 15 SMITHA 0 0 Dec 3 20:37 projecta \$ ===> INPUT ESC=¢ 1=Help 2=SubCmd 3=HlpRetrn 4=Top 5=Bottom 6=TS0 7=BackScr 8=Scroll 9=NextSess 10=Refresh 11=FwdRetr 12=Retrieve	-rw-r	·r	5	SMITHA	0	160	5 Dec	3	16:38	port			
drwxr-xr-x 2 SMITHA 0 0 Nov 17 23:07 src drwxr-xr-x 15 SMITHA 0 0 Dec 3 20:37 projecta \$ ===> INPUT ESC=¢ 1=Help 2=SubCmd 3=HlpRetrn 4=Top 5=Bottom 6=TS0 7=BackScr 8=Scroll 9=NextSess 10=Refresh 11=FwdRetr 12=Retrieve	-rw-r	·r	2	SMITHA	0	47	2 Nov	17	23:15	script			
<pre>drwxr-xr-x 15 SMITHA 0 0 Dec 3 20:37 projecta \$ ===> ESC=¢ 1=Help 2=SubCmd 3=HlpRetrn 4=Top 5=Bottom 6=TS0 7=BackScr 8=Scroll 9=NextSess 10=Refresh 11=FwdRetr 12=Retrieve</pre>	drwxr-x	(r-x	2	SMITHA	0		0 Nov	17	23:07	src			
===> ESC=¢ 1=Help 2=SubCmd 3=HlpRetrn 4=Top 5=Bottom 6=TS0 7=BackScr 8=Scroll 9=NextSess 10=Refresh 11=FwdRetr 12=Retrieve	drwxr-x	(r-x	15	SMITHA	0		0 Dec	3	20:37	projec	ta		
===> INPUT ESC=¢ 1=Help 2=SubCmd 3=HlpRetrn 4=Top 5=Bottom 6=TSO 7=BackScr 8=Scroll 9=NextSess 10=Refresh 11=FwdRetr 12=Retrieve	Ţ												
ESC=¢ 1=Help 2=SubCmd 3=HlpRetrn 4=Top 5=Bottom 6=TS0 7=BackScr 8=Scroll 9=NextSess 10=Refresh 11=FwdRetr 12=Retrieve	===>												
	ESC=¢	1=Не 7=Ва	lp ckSo	2=SubCmd r 8=Scroll		3=HlpRetrn 9=NextSess 1	4=Top 0=Ref	resl	5=Ba n 11=Fi	ottom vdRetr	6=TS 12=Re	0 trieve	

Figure 8: The z/OS shell's display screen after input has been entered

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

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

Working in line mode

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

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

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

Why isn't your output displayed on the screen?

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

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

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

The shell has function keys that you can use for certain tasks, instead of typing commands. To determine your function key settings and escape character assignments, look at the following display:

ESC=¢ 1=Help 2=SubCmd 3=HlpRetrn 4=Top 5=Bottom 6=TS0 7=BackScr 8=Scroll 9=NextSess 10=Refresh 11=FwdRetr 12=Retrieve

The function key functions

Table 1 on page 15 describes all the functions available for function keys, and shows which of those functions are assigned by default to keys. Function keys 13 to 24 are set to the same values as function keys 1 to 12. You can change the default function key settings. For example, you might want a Control function key for typing escape sequences.

You can perform the same actions with either a function key or a subcommand; the term you see under the column Function/Subcommand can be entered as a subcommand also. See <u>"Running a subcommand"</u> on page 22 for more information about subcommands.

In the first column, you see the default key assignment for a function. If a function is not assigned to a key by default, there is no entry in the first column. You can assign a function to a key by customizing your invocation of the OMVS command; see "Function key settings (PFn)" on page 25 for more information.

Default setting	Function or subcommand	Description
<f1> <f13></f13></f1>	HELP	Displays a help panel that explains the TSO/E OMVS command and the three modes you can work in: shell, subcommand, and TSO/E.
<f2> <f14></f14></f2>	SUBCMD	Processes an OMVS subcommand. A subcommand is a command that is passed to the OMVS command processor (instead of the shell). Most subcommands are used to control, or temporarily change, the OMVS interface. You can either enter a subcommand from the shell command line or switch to subcommand mode to do it.
		To run a subcommand from the shell command line, type the subcommand and press this function key.
		To leave the shell session and enter subcommand mode, press this function key when the shell command line is empty. You can type the OMVS subcommands at the command line in subcommand mode. To resume working in the shell, press the Return function key.
<f3> <f15></f15></f3>	HLPRETRN or RETURN	If you are viewing help information, pressing this key removes the help information from the screen. If you are in subcommand mode, pressing this key returns you to the shell session. (Refer to <u>"Switching to subcommand mode" on page 22</u> for a discussion of subcommand mode.)
<f4> <f16></f16></f4>	ТОР	Scrolls displayed data back to a screen of the oldest available output, or, in help, back to the first panel.
<f5> <f17></f17></f5>	BOTTOM	Scrolls displayed data to a screen of the most recent output, or, in help, to the final panel.

Table 1: Function key settings available in the z/OS shell.

Default setting	Function or subcommand	Description
<f6> <f18></f18></f6>	TSO	You have the choice of switching to TSO/E command mode to enter a TSO/E command or running the TSO/E command from the shell command line.
		To switch to TSO/E command mode, press <f6>. To resume working in the shell, press <pa1>.</pa1></f6>
		To run a TSO/E command from the shell command line, select one of these options:
		 Type the command and press the TSO function key.
		- Use the \pm so shell command to run the TSO/E command
		If you entered the OMVS command from ISPF, you cannot enter ISPF as a TSO/E command from the shell command line. You can, however, enter the ISHELL command.
		When the TSO/E command completes, typically *** is displayed on the screen. Press <enter> or <clear> to return to the shell.</clear></enter>
<f7> <f19></f19></f7>	BACKSCR	Scrolls the displayed output backward, a screen at a time. The scrolling ends when you reach the oldest available saved line in a stack of saved output lines.
<f8> <f20></f20></f8>	SCROLL	Scrolls the output display a full screen forward.
<f9> <f21></f21></f9>	NEXTSESS	Displays the next session whose session number is higher than that of the session currently displayed. However, if the highest- numbered session is displayed, the lowest-numbered session will be displayed.
<f10> <f22></f22></f10>	REFRESH	Updates the screen with new data from the shell session. Use this function key if the display of output (for example, output from a command you issued) is incomplete , but the session is now displaying INPUT status.
<f11> <f23></f23></f11>	FWDRETR	Used with <f12> to retrieve commands from the stack of saved input lines. If you press <f12> one too many times and go past the line you want, you can press <f11> to display the line that was previously retrieved by <f12>.</f12></f11></f12></f12>
<f12> <f24></f24></f12>	RETRIEVE	Retrieves the most recently entered input line from a stack of saved input lines. This key starts retrieving with the most recent in the stack of saved lines and works down to the oldest available.
	ALARM	Toggles the setting for the 3270 alarm to sound when the shell writes a <bel> character. Some applications use an alarm to alert the user to particular events. The default setting is to sound the alarm. You can select a key to switch it off and on.</bel>
	AUTOSCR	Toggles the setting for autoscrolling of input and output that is written to the screen. The default setting is to autoscroll; you can select a key to switch it off and on.
	CLOSE	Ends the shell session currently displayed. Close provides the same function as the Quit function key.

Default setting	Function or subcommand	Description
	CONTROL	Treats a character on the command line as part of an escape sequence, and does not append a <newline> character to the sequence. For example, if you type d on the command line and press the Control function key, the system processes the d as the EBCDIC equivalent of the ASCII control sequence <ctrl-d>.</ctrl-d></newline>
	ECHO	Toggles the automatic hiding and display of input. If pressed while an application has control over the display of input, the application no longer controls it. If pressed while the application does not control the display of input, the application is given control.
		See the description of the Hide function key.
	HALFSCR	Scrolls forward half of the currently displayed output.
	HIDE	Toggles the hiding and display of input.
		If you are using OMVS in ECHO mode, pressing this key overrides the visibility asked for by an application, for the next input only. In NOECHO mode, if the input area is not hidden, pressing this key hides the input area for the next input only. If the input area is already hidden, pressing this key makes the input area visible.
	NO	Deactivates a function key so that it performs no function.
	NOALARM	Performs the same function as Alarm.
	NOAUTO	Performs the same function as Auto.
	NOECHO	Performs the same function as Echo.
	NOHIDE	Performs the same function as Hide.
	NOPFSHOW	Toggles the display of function key settings. The default setting is to display the settings; you can select a key to switch the display off and on.
	PFSHOW	Performs the same function as NoPFShow.
	OPEN	Starts another shell session and automatically switches to it. The session is assigned the next unused session number.
	PREVSESS	Displays the next-lower-numbered session. However, if the lowest-numbered session is currently displayed, the highest-numbered session will be displayed.
	QUIT	Ends the current session, and displays the next-lower-numbered session. However, if the lowest-numbered session is currently displayed, the next-higher-numbered session is displayed.
		If only one session is active, Quit causes the OMVS command to quit. The workstation returns to TSO/E, and the shell stops processing.

Table 1: Function key settings available in the z/OS shell. (continued)

Table 1: Function key settings available in the z/OS shell. (continued)

Default setting	Function or subcommand	Description
	QUITALL	Ends all active shell sessions. QuitAll causes the OMVS command to quit. The workstation returns to TSO/E.
		Note: If the OMVS interface is running in SHAREAS mode (shared address space) and you quit all sessions (QuitAll or Quit if there is just one session), the shell process ends immediately.

The escape character

ESC=¢

An escape sequence produces an EBCDIC version of the ASCII control sequence. (For example, the z/OS UNIX <EscChar-D> corresponds to the ASCII <Ctrl-D>.) If you do not use a Control function key to enter escape sequences, you will need to use an escape character. When you type an escape character followed by a second character and press Enter, the second character is converted into a different character before it is passed to the shell.

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

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

ESC=¢`%

The notation EscChar coupled with another letter (for example, <EscChar-D>) indicates an escape sequence.

For more information about escape sequences, see <u>"Typing escape sequences in the shell" on page</u> <u>19</u>.

Entering a shell command

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

Customizing the variant characters on your keyboard

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

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

You may want to customize the encodings for those keys on your keyboard. See <u>Appendix C, "Code page</u> <u>conversion when the shell and MVS have different locales," on page 305</u> for more information on this topic.

Entering a long shell command

If you are typing a long command that will not fit on the command line, you can use the \ (backslash) continuation character at the end of the first line. When you then press <Enter>, the command line is cleared so that you can continue typing. The line you typed prior to the backslash is displayed in the

output area, and beneath it the shell prompt changes to > to indicate that you are continuing a command. For example:

\$ cat /usr/macneil/uts/mydir/mydata\
>
===> /applprog/dbprog/dbget.c
RUNNING

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

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

Entering a shell command from TSO/E

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

oshell shell_command

With this exec, do not use an & to run a shell command in the background. See <u>"OSHELL: Running a shell</u> command from the TSO/E READY prompt" on page 155 for more information.

Interrupting a shell command

If you want to interrupt a command and stop it from completing, type <EscChar-C> or type c and press the Control function key (if you have a function key customized to perform the Control function; see "Determining function key settings and the escape character" on page 15).

Typing escape sequences in the shell

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

- Portable characters not included on your keyboard; see <u>Appendix D</u>, "Escape sequences for a 3270 keyboard," on page 309 for these sequences.
- Control characters that are normally available on ASCII workstations, but not EBCDIC ones; see Appendix D, "Escape sequences for a 3270 keyboard," on page 309 for these sequences.

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

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

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

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

• Type an escape character sequence, beginning with one of the escape characters. After you type the two characters in sequence and press <Enter>, the system translates the two characters into a third character. For information on how to customize your keyboard for typing an escape sequence, see "Keyboard remapping" on page 20.

Suppressing the newline character

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

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

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

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

Keyboard remapping

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

Determining your session status

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

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

The status indicators are:

INPUT

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

RUNNING

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

MORE...

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

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

INPUT HIDDEN

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

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

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

NOT ACCEPTED

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

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

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

NOT ACCEPTED/MORE...

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

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

SUBCOMMAND

Indicates that you are working in subcommand mode.

Scrolling through output

You can scroll output forward and backward using:

- Function keys
- Cursor scrolling
- Scrolling subcommands

Using function keys or subcommands

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

- BackScr
- Bottom
- HalfScr
- Scroll
- Top

These are discussed in Table 1 on page 15.

You can use the TOP, BOTTOM, SCROLL, BACKSCR, and HALFSCR subcommands to scroll output. They produce the same results as their corresponding function keys (see Table 1 on page 15).

Using cursor scrolling

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

BackScr

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

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

HalfScr

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

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

Scroll

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

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

Running a subcommand

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

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

The names of the subcommands match the names of the functions listed in Table 1 on page 15. Some subcommands have aliases; for information on the subcommands and their aliases, see the OMVS command description in *z/OS UNIX System Services Command Reference*.

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

Switching to subcommand mode

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

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

Using multiple sessions

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

Starting sessions

To start additional sessions, you can:

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

Switching between sessions

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

- NextSess is the default setting for <F9>. If you wish, you can customize an additional key for the PrevSess setting. See <u>Table 1 on page 15</u> for a discussion of these functions.
- The NEXTSESS and PREVSESS subcommands perform the same as the function keys.

Customizing the OMVS interface

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

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

An example of customizing the OMVS command

To invoke the OMVS command to:

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

Enter:

omvs pf1(control) sessions(3) noshareas

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

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

The use of the REXX function PROMPT() is required to prevent prompts from being suppressed. Otherwise, TSO/E commands cannot prompt you for additional information when the commands are issued during a shell session.

- 2. Install the exec in a data set that is part of either the SYSPROC or SYSEXEC concatenation.
- 3. When you log on to TSO/E, at the READY prompt you enter MYOMVS and the exec calls MYOMVS, your customized OMVS command. Your changes override the default settings.

For more discussion of the syntax of the OMVS command and its customizable keywords, see the <u>OMVS</u> command description in *z/OS UNIX System Services Command Reference*.

The alarm setting (ALARM | NOALARM)

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

Autoscrolling (AUTOSCROLL | NOAUTOSCROLL)

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

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

The character conversion table (CONVERT)

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

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

Double-byte character set support (DBCS | NODBCS)

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

Use the NODBCS option if you have a DBCS terminal but do not want the overhead associated with using the OMVS command with DBCS support.

Debugging for the OMVS command (DEBUG)

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

Giving an application control of the command line (ECHO | NOECHO)

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

Ending 3270 pass-through mode (ENDPASSTHROUGH)

Applications running from the shell can switch to TSO/3270 pass-through mode, which lets an application invoke TSO/E functions. For application development purposes, you can specify a key that ends TSO/3270 pass-through mode and forces OMVS to return to the shell session.

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

For more information about TSO/3270 pass-through mode, see <u>TSO/3270 passthrough mode</u> in <u>z/OS</u> UNIX System Services Programming Tools.

The escape character (ESCAPE)

If you do not use a Control function key to escape a character, you can type a two-character escape sequence instead. (For an explanation of escape characters, see <u>"Typing escape sequences in the shell"</u> on page 19.)

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

OMVS ESCAPE('`¢')

When specifying escape characters:

- Select characters that are not in the POSIX portable character set. See <u>"The POSIX portable character</u> set" on page 306 to see the contents of the POSIX portable character set.
- Select single-byte characters, even if you are using a double-byte character set.

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

Controlling the size of the output scroll buffer (LINES)

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

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

Function key settings (PFn)

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

OMVS PF1(CONTROL)

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

Displaying the function key settings (PFSHOW | NOPFSHOW)

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

Specifying Language Environment runtime options (RUNOPTS)

To run the TSO/E OMVS command with Language Environment[®] runtime options, specify the RUNOPTS keyword.

Example: To run the OMVS command and print out an options report, issue:

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

See *z/OS Language Environment Programming Reference* for a list of runtime options.

Note: The use of inappropriate Language Environment runtime options, such as TRAP(OFF) or POSIX(OFF), may cause the OMVS command to fail.

Any valid runtime options specified by RUNOPTS normally get passed along to the shell.

Multiple sessions (SESSIONS)

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

The shared TSO/E address space (SHAREAS | NOSHAREAS)

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

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

For more information about shared address space, see <u>Chapter 12</u>, "Performance: Running executable files," on page 161.

Controlling data recorded in the debug data set (WRAPDEBUG)

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

Performing TSO/E work or ISPF work after invoking the shell

After you invoke the shell, you can:

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

Entering a TSO/E command from the z/OS shell

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

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

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

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

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

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

Use the man command to view descriptions of TSO/E commands by prefixing the command with tso. For example, to view a description of the MOUNT command, you would enter:

man tsomount

For complete information about the man command, see <u>man - Display sections of the online reference</u> manual in *z/OS UNIX System Services Command Reference*.

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

Switching to TSO/E command mode

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

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

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

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

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

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

ftp or telnet from TSO

There is an ftp command available in the shell, but no telnet command. However, when you use the OMVS command to login to the shell, you can switch to TSO and issue the telnet command from there, with the following restriction: When you telnet to a remote MVS host and then access a shell, you can work in line mode only (for example, you cannot use vi).

See *z/OS Communications Server: IP User's Guide and Commands* for detailed information about using the telnet command from TSO.

Exiting the shell

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

• To leave the shell temporarily and switch to TSO/E command mode: Press the TSO function key. You can do this any time during a session, regardless of whether you are currently running a command or script. See "Performing TSO/E work or ISPF work after invoking the shell" on page 26 for details.

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

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

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

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

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

• To exit the shell when a background job is running: Press the SubCmd function key and then enter the QUIT subcommand.

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

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

By default in the shell (the set -m option), a background job runs in a different process group from the shell, and the job keeps running after you exit the shell. To have background jobs run in the same process group as the shell, use the set +m command or its equivalent, set +o monitor.

• If your application is in a loop: Try using <EscChar-C> or <EscChar-V> to interrupt it. If this does not work, press the SubCmd function key to leave the shell. Then type quit and press <Enter>. This causes the OMVS command to quit abruptly. The workstation returns to TSO/E and the shell stops processing. For more information on using escape sequences such as <EscChar-C>, see <u>"Typing escape sequences</u> in the shell" on page 19.

Getting rid of a hung application

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

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

Then enter kill -s KILL *nnnnn*, where *nnnnn* is the PID obtained from the ps -ef command. After the kill command completes, you can return to the first session using the NEXTSESS or PREVSESS subcommand.

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

Using a double-byte character set (DBCS)

If you want to display or enter double-byte data, you must:

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

- Access the shell using the OMVS command with the DBCS keyword, the default setting.
- Define a single-byte escape character for typing an escape sequence, if you do not use the default ¢.

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

Single-byte restrictions

When working with a double-byte character set, you must use single-byte characters in these situations:

- Single-byte characters for file names. DBCS characters in file names are treated as SBCS characters.
- Single-byte characters for command-line options
- Single-byte characters for command-line arguments
- Single-byte characters for delimiters, such as a slash, braces, parentheses, and so on
- For user-defined environment variables, only SBCS for the names, and SBCS or DBCS for the values
- For the shell environment variables, only IFS, PS1, and PS2 support DBCS values
- For user IDs, passwords, and password phrases
- For device, group, and terminal names.

Chapter 3. The asynchronous terminal interface to the shells

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

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

ASCII-EBCDIC translation

When you use rlogin, telnet, or Communications Server to access the shell, the data you enter is translated from ASCII (ISO8859-1) to EBCDIC (IBM-1047) before the shell processes it. To change code pages for the current session, use the chcp command. To automatically change code pages after you login, see <u>"Changing the locale in the shell" on page 40</u> for the z/OS shell, or <u>"Changing the locale in the shell" on page 54</u> for the tcsh shell.

For a complete list of the single-byte and double-byte ASCII and EBCDIC code pages that you can specify, see *z/OS XL C/C++ Programming Guide*.

Using rlogin to access the shell

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

To improve performance when you rlogin into a shell, you can use shared address space; for more information, see Chapter 12, "Performance: Running executable files," on page 161.

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

Using telnet to access the shell

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

Using Communications Server login to access the shell

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

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

The shell session

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

When you are in a shell session, you can:

- Run all shell commands and utilities.
- Run any application from the z/OS UNIX file system.
- Use the vi editor and other full-screen applications such as talk and more.

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

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

Entering a shell command

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

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

Interrupting a shell command

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

Using multiple sessions

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

Using a double-byte character set (DBCS)

If you want to display or enter double-byte data:

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

 For information on how to customize your locale and configure your setup files, see "Changing the locale in the shell" on page 40 for the z/OS shell, or "Changing the locale in the shell" on page 54 for the tcsh shell.

When you are working with a double-byte character set, there are some restrictions. See <u>"Single-byte</u> restrictions" on page 29 for more information.

Standard shell escape characters

The following are some of the standard shell escape characters:

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

Chapter 4. Customizing the z/OS shell

If you are interested in working with the z/OS shell, read this information as well as:

- Chapter 6, "Working with z/OS shell commands," on page 61
- Chapter 8, "Writing z/OS shell scripts," on page 107

You can personalize your use of the z/OS shell. This topic covers this information:

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

Customizing your .profile

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

1./etc/profile

- 2.\$HOME/.profile
- 3. The file that the ENV variable specifies.

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

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

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

Typically, your .profile might contain the following lines:

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

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.

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

ENV=\$HOME/.setup export ENV

If portability to a Bourne shell is a consideration, use the two-line syntax. See <u>"Exporting variables" on</u> page 110 for more information about exporting variables.

export ENV=\$HOME/.setup

Identifies the .setup file in your home directory as your login script (also known as a setup script or environment file) and specifies that whenever a shell is created, the ENV variable should be exported to it. See <u>"Customizing your shell environment: The ENV variable" on page 38</u> for more information about a login script.

export PATH=\$PATH:\$HOME:

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

export PATH=\$PATH:\$HOME:.

This PATH setting and the one in the example are equivalent. See <u>"Customizing the search path for</u> commands: The PATH variable" on page 38 for more information.

export PS1='\$LOGNAME:\$PWD: >'

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

turbo:/u/turbo: >

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

export EDITOR=ed

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

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

Quoting variable values

When you have blanks in a variable value, you need to enclose it in 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 (\$, ` (back quote), 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 line:

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, to change the command prompt string to +>, issue:

PS1='+>'

Understanding shell variables

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

set

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

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

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

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



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

echo \$HOME

print \$HOME

printenv \$HOME

displays the current value of the HOME variable.

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

echo *.doc

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

For more information about shell variables, see the <u>sh</u> command description in <u>z/OS UNIX System Services</u> Command Reference.

Customizing your shell environment: The ENV variable

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

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

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

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

export ENV=\$HOME/.setup

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

Customizing the search path for commands: The PATH variable

Command interpreters usually have to search for a file that contains the command you want to run. When you are 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 z/OS shell commands are stored) before the /etc directory.

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

Tip: To ensure that the z/OS shell properly identifies a shell built-in command, specify the shell commands directory /bin exactly as /bin (not as /bin/ or any other variation) in addition to making the shell commands directory /bin part of your PATH. Some commands located in /bin are implemented as shell built-in commands in order to improve performance of shell scripts. The directories specified in PATH influence how the shell locates commands, including the built-in commands, which also influence how the shell handles tracked aliases. See <u>"Using alias tracking" on page 67</u> for more information about tracked aliases.

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 that you have different directories containing the source code for different programs. In each directory, you create a shell script named compile that compiles all the source modules of the program in that directory. To compile a particular program, enter cd to change to the appropriate directory and then enter:

compile

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

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

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

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

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

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

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

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

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

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

Checking the search path used for a command

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

type date

tells you:

date is /bin/date

and the command:

type jobs

tells you:

jobs is a built-in command

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

Customizing the FPATH search path: The FPATH variable

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

Customizing the DLL search path: The LIBPATH variable

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

LIBPATH=/lib:/usr/lib:.

Improving the performance of shell scripts

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

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

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

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

Changing the locale in the shell

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

- "Advantages of a locale compatible with the MVS code page" on page 41
- "Advantages of a locale generated with code page IBM-1047" on page 41
- "Changing the locale setting in your profile" on page 41
- "The LC_SYNTAX environment variable" on page 43
- "The LOCPATH environment variable" on page 44

For additional information about locale and LC_SYNTAX, see <u>z/OS Language Environment Programming</u> *Guide*.

Advantages of a locale compatible with the MVS code page

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

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

Customizing for a locale not based on code page IBM-1047

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

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

Advantages of a locale generated with code page IBM-1047

You might prefer using one of the locales that is compatible with IBM-1047, but not compatible with the MVS code page if:

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

There are other important code page conversion considerations when the shell uses code page IBM-1047 and MVS does not; see <u>Appendix C, "Code page conversion when the shell and MVS have different</u> locales," on page 305 for that information.

Changing the locale setting in your profile

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

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

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

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

If you place an export LC_ALL=*localename* statement in your login profile, or if one has been placed in /etc/profile, make sure it is followed with exec sh -L and protect that with tty -s, as shown in <u>"Examples: Changing locale" on page 42</u>. If you don't protect it with the tty -s test, BPXBATCH SH *command* will not run the command.

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

- 1. Loop control; you only want the exec sh -L to be executed the first time.
- 2. If you plan to use BPXBATCH or OSHELL (which calls BPXBATCH) with national language support, you need to define the LANG and LC_ALL variables in a file for BPXBATCH to use. See <u>"Passing</u> environment variables to BPXBATCH" on page 147 for more information.

If your /etc/profile has been set up for the proper locale, you only need to change your .profile if you want a different locale than already set up as the default. For more information on setting up locale and messages, see the section on customizing for your national code page in the shell in <u>z/OS UNIX</u> System Services Planning.

Examples: Changing locale

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

```
if test -z "$LOCALE_SWITCH" && tty -s
then
 echo "
                                 - - - - - - - -
 _ 0
 echo " - Logon shell will now be invoked to reflect
                                       - - - - - - - - - - "
 LOCALE_SWITCH=EXECUTED
 LANG=C
 LC ALL=Da DK.IBM-277
 export LANG LC_ALL LOCALE_SWITCH
#Issue chcp if not using OMVS command
if test "$_BPX_TERMPATH" ! "OMVS"
   then
  chcp -a IS08859-1 -e IBM-277
  fi
exec sh -L
else
 echo "
        - - - - - -
                                 . . . . . . . . . . . .
 echo " - Welcome to OS/390 UNIX System Services
 echo " - - - -
fi
```

If you want your messages displayed in a different language than that specified in the system-wide /etc/ profile, you must modify your .profile accordingly. For more information, see <u>"Customizing the</u> language of your messages" on page 44.

For a list of the z/OS UNIX locales (and their locale object names) and locale source files, see <u>Appendix E</u>, "Locale objects, source files, and charmaps," on page 315.

The LC_SYNTAX environment variable

There are 13 variant characters in the POSIX portable character set whose encoding might vary on different EBCDIC code pages:

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

Before MVS SP Release 5.2.2, the z/OS shell and utilities required that all data in the z/OS UNIX file system be encoded in one of three code pages: IBM-1047, IBM-1027, or IBM-939. Any data moved into the z/OS UNIX file system from a workstation or from an MVS data set often had to be converted to one of code pages IBM-1047, IBM-1027, or IBM-939 before it could be processed by the shell. Similarly, to ensure that any variant characters keyed in at the terminal had the correct encoding, you had to either use the conversion option of the OMVS command or customize your keyboard.

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

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

LC_SYNTAX—an example

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

echo 'this is\nreal handy'

produces the following output at the terminal:

```
this is
real handy
```

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

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

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

Limitations

The LC_SYNTAX setting does not affect:

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

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

If shell scripts are tagged and automatic conversion is not enabled, then the code page in which a shell script is encoded must match the code page of the locale in which it is run.

If shell scripts are tagged and automatic conversion is enabled, then the locale must indicate a SBCS code page and the scripts must be SBCS.

The LOCPATH environment variable

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

Customizing the language of your messages

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

export LANG=your_language

your_language is the first part of the locale name listed in <u>Appendix E</u>, "Locale objects, source files, and <u>charmaps</u>," on page 315. For example, Ja_JP in the locale name JA_JP.IBM-939. Currently, z/OS UNIX ships messages in English, Kanji, and Simplified Chinese.

Setting your local time zone

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

The three primary fields in the time zone specification are:

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

Example: If you want to set your time zone to Eastern Standard Time (EST) and export it, specify:

export TZ="EST5EDT"

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

For complete information about how to specify the local time zone, see Format of the TZ environment variable in *z/OS UNIX System Services Command Reference*.

Building a STEPLIB environment: The STEPLIB environment variable

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

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

You can set the variable in one of three ways:

Statement	Action
STEPLIB=CURRENT	Passes on any currently active TASKLIB, STEPLIB, or JOBLIB allocations from the invoker's MVS program search order environment to the environment created for the executable file to run in. Any STEPLIB environment in the invoker's process image is recreated in the new process image for the executable file when the file is invoked. This is the default value that is set if no STEPLIB variable is specified. If an application uses fork(), spawn(), or exec(), the STEPLIB data sets must be cataloged.
STEPLIB=NONE	Specifies that no STEPLIB environment should be set up for executable files.
STEPLIB=DSN1:DSN2:DSN3	Sets up a library search order for the STEPLIB, in the order that the data sets are specified. You can specify up to 255 fully qualified data set names, separated by colons. For example:
	export STEPLIB=SMITH.C.LOADLIB:SMITH.PL1.LOADLIB
	The specified data sets must be cataloged MVS load libraries that you have security access to. The data sets specified here are built into a STEPLIB environment for the executable file.

Table 2: Three ways to set the STEPLIB environment variable (z/OS shell)

Restrictions on STEPLIB data sets

For executable files that have the set-user-ID or set-group-ID bit set, there are restrictions on the data sets that can be built into the STEPLIB environment for the file to run in. The systems programmer maintains a STEPLIB sanction list of data sets that can be included in the STEPLIB environment for such executable files. Only data sets on that list are built into the STEPLIB environment for such files. If you need a data set added to the list, contact your systems programmer.

Setting options for a shell session

The set command lets you set options, or flags, for your shell session. These flags control the way that the shell handles certain situations.

1. To display the shell flags that are currently set, enter:

set -o

2. To turn on an option, 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 that is specified on the ENV variable).

3. To turn off an option, 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 set command description in *z/OS UNIX System Services Command Reference*.

Exporting variables

The command:

set -o allexport

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

Controlling redirection

The command:

set -o noclobber

indicates that you do not want the > redirection operator to overwrite existing files. When this option is on and you specify the construct >*file*, the redirection works only if *file* does not 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 "<u>"Using a wildcard character to specify file names" on page 74</u>" 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 74 for a discussion of wildcard characters.

Displaying input from a file

The command:

set -o verbose

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

Running a command in the current environment

The command:

set -o pipecurrent

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

Displaying current option settings

The command:

set -o

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

- o to indicate that the option is enabled

+o to indicate that the option is disabled

Chapter 5. Customizing the tcsh shell

If you are interested in using the tcsh shell, read this information as well as:

- Chapter 7, "Working with tcsh shell commands," on page 85
- Chapter 9, "Writing tcsh shell scripts," on page 125

You can personalize your use of the tcsh shell. This topic covers these tasks:

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

Understanding the startup files

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

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

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

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

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

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

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

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

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

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

```
# set TERM environment variable
setenv TERM vt220
# set DISPLAY environment variable
setenv DISPLAY mymachine.mydomain.com:0
```

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

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

Quoting variable values

When you have blanks in a variable value, you need to enclose it 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:

setenv HOMEMSG "Using \$HOME as Home Directory"

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

echo \$HOMEMSG

would display:

Using /u/user as home directory

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

setenv HOMEMSG 'Using \$HOME as home directory'

the following:

echo \$HOMEMSG

would display:

Using \$HOME as home directory

As you can see, the \$ is not expanded.

Changing variable values dynamically

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

To change the command prompt string to +>, issue:

set prompt='+>'
Understanding shell variables

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

set

or

set -r

set -r displays readonly shell variables.

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

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

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

setenv printenv

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

echo \$HOME

printenv \$HOME

displays the current value of the HOME variable.

In general, echo displays the current values of all its arguments, after any shell processing has taken place. The shell first expands the wildcard character *.

Example: Consider:

echo *.doc

Result: This produces the names of every file in the working directory that has the suffix **.doc**. So the output of echo is a list of all such files. And if there are no file names ending in **.doc**, the command output is just *.doc.

For more information about shell variables, see the <u>tcsh</u> command description in <u>z/OS UNIX System</u> Services Command Reference.

Customizing your shell environment: The .tcshrc file

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

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

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

Following is a sample .tcshrc file:

```
‡Ł
           path shell variable
ŧ
# Lists directories in which to look for executable commands.
#set path = ( /bin /usr/local/bin /usr/bin )
# test if we are an interactive shell
if ($?prompt) then
₫Ŀ
           prompt shell variable
ŧ
# The string which is printed before reading each command from the
# terminal. Currently set to display hostname, and current working
# directory.
# ______
set prompt = "%m:%~> "
rmstar shell variable
₫Ŀ
‡ŧ
# If set, the user is prompted before 'rm *' is executed.
set rmstar
‡ŧ
          noclobber shell variable
# If set, output redirection will not overwrite existing files.
#set noclobber
# source complete.tcsh
if (`filetest -e /etc/complete.tcsh`) then
   source /etc/complete.tcsh
endif
endif # interactive shell
# _____
# set up useful aliases
alias m more
```

Figure 9: A sample .tcsh file

Customizing the search path for commands: The PATH variable

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

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

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

or,

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

For example, you might enter:

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

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

1./bin

```
2./usr/bin
```

- 3./usr/macneil/bin
- 4./usr/games
- 5./usr

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

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

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

Adding your working directory to the search path

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

compile

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

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

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

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

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

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

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

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

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

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

Checking the search path used for a command

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

which kill

tells you:

kill: shell built-in command.

and the command:

where kill

tells you:

kill is a shell built-in /bin/kill

Customizing the DLL search path: The LIBPATH variable

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

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

Changing the locale in the shell

The default locale for the shell and utilities is C.

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

Advantages of a locale compatible with the MVS code page

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

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

Customizing for a locale not based on code page IBM-1047

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

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

Advantages of a locale generated with code page IBM-1047

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

• You already use one of the IBM-1047 locales and have made an investment in data conversion and keyboard remapping.

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

There are other important code page conversion considerations when the shell uses code page IBM-1047 and MVS does not; see <u>Appendix C, "Code page conversion when the shell and MVS have different</u> locales," on page 305 for that information.

Changing the locale setting in your profile

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

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

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

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

If you place a setenv LC_ALL *localename* statement in your login profile, or if one has been placed in /etc/csh.login, make sure it is followed with exec tcsh -l and protect that with tty -s, as shown in <u>"Examples: Changing locale" on page 55</u>. If you don't protect it with the tty -s test, BPXBATCH SH *command* will not run the command.

If you use exec tcsh -1, there are two situations that you must take into account:

- 1. Loop control; you only want the exec tcsh -1 to be executed the first time.
- 2. If you plan to use BPXBATCH or OSHELL (which calls BPXBATCH) with national language support, you need to define the LANG and LC_ALL variables in a file for BPXBATCH to use. See <u>"Passing</u> environment variables to BPXBATCH" on page 147 for more information.

If your /etc/csh.login was set up for the proper locale, you only need to change your .login if you want a different locale than already set up as the default. For more information on setting up locale and messages, see the section on customizing for your national code page in the shell in <u>z/OS UNIX System</u> Services Planning.

Examples: Changing locale

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

```
setenv LANG C
setenv LC_ALL Da_DK.IBM-277
# Issue chcp if not using OMVS command
if ($?_BPX_TERMPATH != "OMVS" ) then
chcp -a IS08859-1 -e IBM-277
endif
exec tcsh -l
endif
unset tty rc
```

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

For a list of the z/OS UNIX locales (and their locale object names) and locale source files, see <u>Appendix E</u>, "Locale objects, source files, and charmaps," on page 315.

The LC_SYNTAX environment variable

There are 13 variant characters in the POSIX portable character set whose encoding might vary on different EBCDIC code pages:

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

When you specify a locale with the LC_ALL variable, the LC_SYNTAX environment variable is set. The shell uses the LC_SYNTAX environment variable to determine the code points to use for the 13 variant characters. This means that the shell can dynamically adapt to the code page of the current locale.

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

LC_SYNTAX—an example

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

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

produces the following output at the terminal:

this is real handy

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

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

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

Limitations

The LC_SYNTAX setting does not affect:

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

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

If shell scripts are tagged and automatic conversion is not enabled, then the code page in which a shell script is encoded must match the code page of the locale in which it is run.

If shell scripts are tagged and automatic conversion is enabled, then the locale must indicate a SBCS code page.

The LOCPATH environment variable

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

Customizing the language of your messages

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

setenv LANG your_language

your_language is the first part of the locale name listed in <u>Appendix E, "Locale objects, source files, and charmaps," on page 315</u>. For example, Ja_JP in the locale name JA_JP.IBM-939. Currently, z/OS UNIX ships messages in English, Kanji and Simplified Chinese.

Setting your local time zone

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

The three primary fields in the time zone specification are:

- 1. The local standard time, abbreviated—for example, EST or MSEZ.
- 2. The time offset west from the universal reference time, typically specified in hours (minutes and seconds are optional). A minus sign (-) indicates an offset east of the universal reference time.

3. The daylight saving time zone, abbreviated—for example, EDT. If this and the first field are identical or this value is missing, daylight saving time conversion is disabled. Optionally, you can specify an additional rule that indicates when daylight saving time starts and ends.

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

setenv TZ "EST5EDT"

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

For complete information about how to specify the local time zone, see Format of the TZ environment variable in *z/OS UNIX System Services Command Reference*.

Building a STEPLIB environment: The STEPLIB environment variable

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

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

You can set the variable in one of three ways:

Statement	Action
setenv STEPLIB CURRENT	Passes on any currently active TASKLIB, STEPLIB, or JOBLIB allocations from the invoker's MVS program search order environment to the environment created for the executable file to run in. Any STEPLIB environment in the invoker's process image is re-created in the new process image for the executable file when the file is invoked. This is the default value that is set if no STEPLIB variable is specified.
	If an application uses fork(), spawn(), or exec(), the STEPLIB data sets must be cataloged.
setenv STEPLIB NONE	Specifies that no STEPLIB environment should be set up for executable files.
setenv STEPLIB DSN1:DSN2:DSN	Sets up a library search order for the STEPLIB, in the order that the data sets are specified. You can specify up to 255 fully qualified data set names, separated by colons. For example:
	setenv STEPLIB SMITH.C.LOADLIB:SMITH.PL1.LOADLIB
	The specified data sets must be cataloged MVS load libraries that you have security access to. The data sets specified here are built into a STEPLIB environment for the executable file.

Table 3: Three ways to set the STEPLIB environment variable (tcsh shell

Restrictions on STEPLIB data sets

For executable files that have the set-user-ID or set-group-ID bit set, there are restrictions on the data sets that can be built into the STEPLIB environment for the file to run in. The systems programmer

maintains a STEPLIB sanction list of data sets that can be included in the STEPLIB environment for such executable files. Only data sets on that list are built into the STEPLIB environment for such files. If you need a data set added to the list, contact your systems programmer.

Setting variables for a shell session

With the set and unset commands, you can set and unset variables for your shell session. These variables control the way that the shell handles certain situations. To display the shell variables that are currently set, type set . To turn on an option, enter the following command:

set name

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

To turn off an option, enter:

unset name

The following discussion highlights some of the options that you might find useful. For all the options, see the information about set in the tcsh shell in the <u>set</u> command description in <u>z/OS UNIX System Services</u> Command Reference.

Displaying current option settings

The command:

set

displays all current option settings.

Controlling redirection

The command:

set noclobber

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

Preventing wildcard character expansion

The command:

set noglob

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

Displaying input from a file

The command:

set xtrace

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

Displaying deletion verification

The command:

set rmstar

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

Files accessed at termination

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

1./etc/csh.logout

2.\$HOME/.logout

Chapter 6. Working with z/OS shell commands

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

Specifying shell command options

Most of the commands discussed in this topic accept options. Shell command options are usually specified by a minus sign (–) followed by a single character. For example, the 1s 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 <u>"Listing directory</u> contents" on page 195 for an example.)

ls -1

lists directory names in a single column.

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

ls -1 dir1

to list the contents of dir1 in a single column.

Command options and arguments must be typed as single-byte characters. Additionally, delimiters such as a slash, braces, and parentheses must be typed as single-byte characters.

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

ls dir1 -F

1s 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 z/OS shell commands let you specify a double minus sign (--) to separate the options from the nonoption arguments; - - means that there are no more options. Thus, if you really have a directory named –F, you could enter:

ls -- -F

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

The z/OS 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 <u>cat</u> command description in <u>z/OS UNIX System Services Command</u> *Reference*.) You could enter:

cat -t -v file

or you could combine the two options into:

cat -tv file

The order of the options is not important:

cat -vt file

is equivalent to the previous version of the command.

Specifying options with accompanying arguments

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

For example, if you read the <u>ps</u> command description in <u>z/OS UNIX System Services Command Reference</u>, you will see that ps accepts an argument of the form:

-u userlist

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

ps -u 'macneil,wellie1'

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

Help for shell command usage

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

For online help information about a command, see <u>"Using the man command to get online help" on page</u> 82.

Understanding standard input, standard output, and standard error

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

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

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

Using the shell: In the shell, the names for these files are:

- stdin for the standard input file.
- **stdout** for the *standard output* file.
- stderr for the standard error file.

The shell sometimes refers to these files by their *file descriptors,* or identifiers:

- 0 for stdin
- 1 for stdout
- 2 for stderr

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

Using TSO/E: When you are invoking the BPXBATCH utility, you can specify these standard files in MVS DD statements, TSO/E ALLOCATE commands, or DYNALLOC macros using the ddnames:

- STDIN for standard input
- STDOUT for standard output
- STDERR for standard error

For more information about BPXBATCH, see "The BPXBATCH utility" on page 145.

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

Redirecting command output to a file

Commands entered at the command line typically use the three standard files described previously, 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 z/OS 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:

>>file name

instead. For example:

sort -u file1 >output 2>>outerr

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

Suppose you entered:

sort -u filea 2>&1 >output

In this command, you see two redirections:

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

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

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

Redirecting input from a file

You can redirect input in much the same way that you redirect output. A command that normally takes input from standard input can be redirected to take input from a file instead.

To send the file power to another user, issue:

mailx DEEJ <power</pre>

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

Redirecting error output to a file

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

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

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

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

If you do not care about seeing the error output, you can 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*, your workstation screen, in several different formats.

od file

dumps a file in octal.

od -h file

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

Setting up an alias for a command

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

alias name="string"

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

For example, suppose you have a hard time remembering that the mv command actually renames files. Instead, 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

Tip: If you issue an exec sh, alias names are not exported. For information about how to put alias definitions in your login script pointed to by the ENV variable, see <u>"Customizing your shell environment:</u> The ENV variable" on page 38.

DBCS recommends that you use single-byte characters when specifying an alias name, because the POSIX standard states that alias names must contain only characters in the POSIX portable character set.

Defining an alias

If you will be using an alias frequently, put the alias command in your profile file (**\$HOME/.profile**). When you issue the OMVS command or start a shell with sh -L, the shell reads the aliases from the file and sets them up immediately. That way, you do not have to type them in every time you start using the shell. See <u>"Customizing your .profile" on page 35</u> for more information about customizing your profile file.

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

alias

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

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

alias functions="typeset -f"

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

alias f=functions

Then you enter:

f abc

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

"typeset -f"

Redefining an alias for a session

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

alias name="string"

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

Setting up an alias for a particular version of a command

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

grep hello file

displays all the lines of *file* that contain the string hello. Normally, grep distinguishes between uppercase and lowercase letters; this means, for example, that the search in the previous example does not display lines that contain 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 z/OS shell checks specifically for an alias of the same name, and does the correct thing.

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

Using alias tracking

Alias tracking can reduce the time the shell spends searching your search path (specified with the **PATH** variable) for a command; it helps shell scripts run faster. A *tracked alias* is a shell-created alias that is the full pathname for a command. The shell automatically tracks everything it finds in the default path for executables (**/bin**). For example, if you enter the ps command, the shell creates the alias:

ps="/bin/ps"

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

```
set -o trackall
```

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

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

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

To list your tracked aliases, enter the command:

alias -t

To turn off alias tracking of all commands, enter the command:

set +o trackall

Then commands found in directories other than **/bin** are not tracked. When the **PATH** search finds a command in **/bin**, the pathname will always be tracked.

To remove tracked aliases, use:

alias -r

Turning off an alias

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

• Get rid of the alias entirely. The command:

unalias rm

gets rid of the rm alias for the session. After this, when you enter rm, you get the real rm command.

• Escape the alias. If you put a backslash in front of an alias, the shell uses the real command rather than the alias. For example:

\rm file

• Specify the full pathname. For example:

/bin/rm file

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

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

Combining commands

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

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

Using a pipe (|) or a filter with a pipe

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

Using a semicolon (;)

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

cd mydir ; ls

Also, if you have defined the alias:

alias l="ls -l"

you can enter:

cd mydir ; l

because you can use aliases such as 1 after a semicolon.

Using && and ||

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

&&

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

Ш

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 z/OS 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 1s to obtain information about 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 of your processes that are currently active in the system and 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 it encounters a construct of the form:

\$(command)

or:

`command

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

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

As an example of how a programmer could use command substitution, consider a file called **srclist**, containing the following list of source code 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 1s 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 1s information only for files that were last changed in November. (To be perfectly accurate, it also displays information about files that have the string Nov in their names, too.)

Another useful find option has the form:

find path -ctime number

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

ls -l \$(find dir -ctime 1)

displays 1s information about 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 z/OS UNIX find command automatically prints its results without – print. However, if you have an existing shell script or compatibility with UNIX systems is important to you, you can use –print.

For more information about the find command, see the find command description in *z/OS UNIX System Services Command Reference*.

Characters that have special meaning to the shell

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

Characters used with commands

Character

Usage

Pipes the output from one command to a second command; separates commands in a *pipeline*.

Ш

L

Separates two commands. If the command preceding || fails, it runs the following command (Boolean OR operator).

&

Runs a command in the background, if placed at the end of a command line.

Used in redirection, &0 represents standard input, &1 represents standard output, and &2 represents standard error.

&&

Separates two commands. If the command preceding && succeeds, it runs the following command (Boolean AND operator).

;

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

()

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

(s_p) is also used to group mathematical operations.

{}

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

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

#

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

\$

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

١

The backslash character turns off the special meaning of the character that follows it. For more information, see <u>"Using a special character without its special meaning" on page 73</u>.

• •

A pair of single quotation marks turns off the special meaning of all characters within the quotes. For more information, see <u>"Using a special character without its special meaning" on page 73</u>.

....

A pair of double quotation marks turns off the special meaning of the characters within the quotes, except for \$, `, ", and $\$. See <u>"Using a special character without its special meaning" on page 73</u> for more information.

Characters used in file names

Character Usage /

Separates the components of a file's pathname.

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

~susanb/.profile

refers to user SUSANB's .profile file.

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

cd ~-

returns you to the directory you were previously working in.

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

••

•

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

?

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

*

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

Character	Usage	Example
<	Redirects input to a specified file.	"Redirecting input from a file" on page 64.
>	Redirects output to a specified file.	"Redirecting command output to a file" on page 63.
>>	Redirects output to be appended to the end of the specified file.	"Redirecting command output to a file" on page 63.
2>	Redirects error output to a specified file.	"Redirecting error output to a file" on page 64.

Redirecting input and output

Character	Usage	Example
< <text <i="" encounters="" input="" it="" reads="" standard="" until="">text.</text>	Reads standard input until it encounters <i>text</i> .	This is used in what is called a "here-document." Input is usually typed on the screen or in a shell script. For example, this script creates a file called hello.c , compiles it into hello , and then executes it:
	<pre>echo "Creating program source" if cat > hello.c <<end_of_file "compiling="" "executing="" #="" \$?="" .="" <="" cat="" echo="" else="" end_of_file="" exec="" exit="" failed="" fi="" hello="" if="" main()="" make="" pre="" program"="" puts("hello,="" then="" world!");="" {="" }=""></end_of_file></pre>	

When you run the shell script, it runs the cat > hello.c command using the input between the two End_of_File strings.

Using a special character without its special meaning

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

\ ...

The backslash

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

echo it\'s me

prints:

it's me

If you just try:

echo it's me

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

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

A pair of single quotation marks (' ')

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

A pair of double quotation marks (" ")

A pair of double quotation marks (" ") turns off the special meaning of the characters within the quotes, 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*. (In an MVS context, the wildcard character is referred to as a *global character*, or *pattern-matching character*.) A wildcard character is a special character that may be used to save typing in file names in shell commands. The z/OS shell recognizes several different wildcard characters:

```
*
?
[]
```

The * character

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

ls aa*

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

The command:

mv *.c dir1/dir2

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

You can use the * wildcard character in directory names as well as in 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 pathname, the question mark ? can stand for any single character, except a leading dot. For example:

file.?

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

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

ls *.?

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

The square brackets

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

[bch]at

matches bat, cat, or hat.

ls [abc]*

lists all files in the working directory the names of which start with a, b, or c, followed by any other sequence of zero or more characters. In other words, it lists all files whose names start with a, b, or c.

You can specify ranges of characters inside the square brackets by specifying the first character in the sequence, a hyphen (-), and the last character. For example:

[a-m]

This matches any character from a through m.

Suppose, for example, that you want to copy the contents of the working directory into two separate directories. You might enter:

```
cp [a-m]* dira
```

to copy all files with names beginning with the letters a through m to the directory dira, and then issue the second command:

cp [n−z]* dirb

to copy the rest of the files to the directory dirb. A command such as:

rm *.[a-z]

removes every file with a suffix consisting of a single lowercase letter.

If the first character inside a bracket construct is an exclamation mark !, the construct matches any character that *is not* inside the brackets. For example:

ls [!a-m]*

lists any file that does not begin with one of the letters in the range a through m.

In the same way:

rm [!0-9]*

removes any file with a name that does not start with a digit.

Retrieving previously entered commands

In the shell, you can retrieve previously issued commands using:

- The history command, combined with the r command
- The two retrieve function keys that are part of the TSO/E OMVS command interface to the shell
- · Command-line editing, when you are using an asynchronous terminal interface

Retrieving commands from the history file

The shell records each command that you enter in a file under your home directory. This file is called the *history file*; its name is .sh_history.If you enter the command:

history

the shell displays the current contents of your history file. Each command is numbered.

You can rerun any of the commands in your history file by typing r, followed by a space, followed by the number of the command you want to use. Think of r as the redo command.

For example, suppose that you are a programmer and you enter a complicated command to compile part of a program. The program contains a syntax error, so you call a text editor to edit the source code and correct the problem. Now you want to run the same compile command on the corrected program. You might save yourself a good deal of typing by using:

history

to find out the number of the previous compile command; you can then run the command with r.

Another time-saver is to specify your shell prompt as:

PS1='(!)\$'

in your .profile. The shell prompt is then preceded by the number assigned to the command in the command history file.

This is how you use the command numbers to enter a command. To repeat command number 14, enter:

r 14

The shell displays the original command 14 in the output area of the screen and then runs it. If you get another error, you can correct it, and then compile again with another r 14. You can perform the operation many times, but you have to type the original only once.

If you type **r** followed by a space, followed by a string of characters (not beginning with a digit), the shell checks backward through the history file and runs the most recent command that begins with the given string. For example, let's look at the compilation example. Suppose that you are using the c++ command to compile your program. Then:

r c++

looks back through the history and runs the most recent c++ command. You do not even have to check on the number of the command you want to enter. The shell displays the selected command in the output area of the screen and then runs it.

This backward-search feature of r can search for aliases as well as normal commands. r searches for the beginning of the command line as you typed it, not the way that the line looked after the alias was replaced.

If you enter r without a number after it, the shell repeats the most recent command.

Editing commands from the history file

Suppose that you have a sequence of source files named **file1.c**, **file2.c**, **file3.c**, and so on that you want to compile with similar c89 commands. This situation is a little different from the one discussed in the previous topic. You do not want to rerun the *same* command for each file; the command has the same form each time, but you have to specify in a new file name each time.

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

r old_string=new_string command

runs a previous *command* but replaces the first occurrence of the *old* string with the *new* string. For example, suppose you compile **file1.c** with:

c89 options file1.c

Then the command:

r file1=file2 c89

tells the shell to search back for the most recent c89 command and 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.

Entering alias displays all the currently defined aliases. You will see a number of aliases that you didn't set up; for example:

history="fc -l"

The history command is actually a *predefined* alias for the fc command with the -l option. The fc command is used to display and edit commands in the history file. Generally, it is easier to remember to type history, so the shell predefines this alias.

If you have displayed the predefined aliases, you probably noticed that r is also a predefined alias. It also stands for a version of the fc command. As with history, the r alias was created because it's easier to use and read than the straight fc command. For full details about fc, see the <u>fc</u> command description in *z/OS UNIX System Services Command Reference*.

Using the retrieve function keys

When you are using the OMVS interface, there are two function key settings for retrieving commands:

Retrieve

This key performs a "backward retrieve" function. It retrieves a saved command from a stack of saved input lines, starting with the most recent and moving down to the oldest available line.

FwdRetr

This key is used with the Retrieve key to retrieve commands from the stack of saved input lines. If you press the Retrieve key one too many times and go past the line you want, you can press the FwdRetr key to display the line that was previously retrieved by the Retrieve key.

Press the Retrieve key repeatedly until the command you want to use is displayed on the command line. Once the command is displayed, you can modify the command or use it as it is displayed. Press <Enter> to run the command.

Command-line editing

When you use rlogin or telnet to login to the shell, you can use command-line editing. Command-line editing lets you access commands from your history file, edit them, and run the result. You have already seen this process before, when reading about some of the features of the r command.

Command editing is useful at those times when you are running the same sequence of commands, or slight variations on the same sequence of commands. The point of command editing is to save yourself the trouble of typing the same thing over and over again—look especially for long commands that normally require a lot of typing. Command editing is also useful when you have made a mistake in typing a command line and wish to correct it.

Using the vi command editor

If you run the command:

set -o vi

or

export EDITOR=vi

it tells the shell that you want the ability to edit commands the way that you normally edit text with vi; you are set up for vi command editing. Whenever the shell prompts you for input, it is as if the shell puts you into vi insert mode on a new line at the end of the history file. You can type in a new command just as you normally would.

You can also press <Esc> to enter a vi-like command mode. When you enter command mode, you can use the usual cursor movement commands to move around on the command line, or to move up and down in the history file. For example:

- Press the k key to move back to the previous line in the history file (the last command line you entered). Press the k key again, and you move to the line before that.
- Press j and you move forward in the history file.

In this way it is simple to retrieve recent commands from the history file. You can then edit them using standard vi commands. For example, you can use \$ to move to the end of the line, and A to begin appending text to the end of the line. When you have edited the line to produce the command that you want to run, simply press <Enter> to run that line.

As you might expect, you can use these search commands:

/string ?string

to search backwards and forwards through the history file. You can edit the command line with these vi commands:

w

Move to next word

b

Move to previous word

d

delete

С

change

a

append

i

insert

u

undo

and many of the other vi commands. For a complete list of available commands, see the <u>shedit</u> command description in *z/OS UNIX System Services Command Reference*.

Using the emacs command editor

To set up for emacs command editing, enter:

set -o emacs

This lets you use commands identical to emacs commands to edit your shell command line. For more information, see the description of shedit in *z/OS UNIX System Services Command Reference*.

Using record-keeping commands

Record-keeping commands can be very helpful for programmers. For example, suppose you have a program that is split into several source files. For the sake of simplicity, assume that the source files all have the extension **.c** and are all stored in a subdirectory called src. (To read about extensions, see "Naming files" on page 199.)

It is often the case that you want to find out which source files in the subdirectory refer to a particular variable or function. You can do this very simply with the command:

```
grep 'name' src/*.c
```

The command checks all the appropriate files in the subdirectory src and displays the lines that contain the given *name*. Each line is labeled with the name of the file that contains the line. You can quickly find the use of a function or data object in source files.

As another example of using record-keeping commands, suppose that you are working on a large program and every few days you back up the source code for the program by copying it to a directory in a different file system (as a precaution). You would like to compare the current version of your source files with one of the saved versions, to find out what changes have been made between the two. The command:

diff oldfile newfile

prints out all the differences between two versions of a file, making comparisons possible.

The cksum command gives a checksum for each file. If applied to two versions of what was at one time the same file, cksum gives a convenient way to tell if the files are still the same. It does not, however, indicate what the differences are.

The find command also has applications to programming. For example, suppose you are looking for a particular C source program but cannot remember where it is stored.

find / -name '*.c'

searches all the files and file systems, starting at the root, and displays the names of all files with the .c extension.

Finding elements in a file and presenting them in a specific format

awk is a powerful command that can perform many different operations on files. The general purpose of awk is to read the contents of one or more files, obtain selected pieces of information from the files, and present the information in a specified format.

One simple way to use awk is with a command line with the form:

awk '/regexp/ {action}' file

This asks awk to obtain information from the specified file. awk obtains the information by performing the specified *action* on every line in the file that contains a string matching the given regular expression, *regexp*. (For further information, see <u>Regular expressions (regexp)</u> in <u>z/OS UNIX System Services</u> Command Reference.) For example:

awk '/abc/ {print}' file

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

For more discussion on using awk, see Appendix B, "Using awk," on page 281.

Timing programs

The time command lets you time programs to find out how much processor time they actually require. You might use this to compare two versions of a program to see if one runs faster than the other. You can run a program with:

time command-line

where *command-line* is a command line that invokes the program you want to time. time runs the program and displays:

- The total time the program took to execute, labeled real
- The total time spent in the user program, labeled user
- The central processor time spent performing system services for the user, labeled sys

For more information, see the time command description in *z/OS UNIX System Services Command Reference*.

Using the passwd command

You can change a user's password or password phrase by using the passwd command:

passwd [-u userid]

The passwd command changes the login password or password phrase for the user ID specified. If *userid* is omitted, the login name associated with the current terminal is used. You are prompted for the new password or password phrase.

For example:

passwd

changes the password or password phrase for the invoker. The invoker is prompted for the old password or password phrase and then for the new value.

Non-superusers can change the password or password phrase for another user if they know the user ID and the current password or password phrase. This example changes the password or password phrase for user ID **steve**:

passwd -u steve

For more information about the passwd command, see the <u>passwd</u> command description in <u>z/OS UNIX</u> <u>System Services Command Reference</u> For information on setting up RACF to enable password phrase support, see <u>z/OS Security Server RACF Security Administrator's Guide</u>.

Switching to superuser or another ID

With the su command, you can switch to any user ID, including the superuser. A user can switch to superuser authority (with an effective UID of 0), if the user is permitted to the BPX.SUPERUSER resource in the FACILITY class within the Resource Access Control Facility (RACF). Either the ISPF shell or the su shell command can be used for switching to superuser authority.

If you do not specify a user ID, the su command changes your authorization to that of the superuser. If you specify a user ID, su changes your authorization to that of the specified user ID.

When you switch to superuser (UID 0) without specifying a user ID, you keep your MVS identity (TSO/E ID). You keep your access authority to MVS data sets, while you are gaining authority to access any files.

When you change user ID by specifying a user ID and password or password phrase, you assume the MVS identity of the new user ID even if the user ID has UID 0.

If you use the -s option on the su command, you are not prompted for a password. Use this option if you have access to the BPX.SRV.*userid* SURROGAT class profile. The *userid* is the MVS user ID associated with the target UID.

To return to your own user ID, type:

exit

You go back to the shell in which you entered the su command.

For more information, see the su command in *z/OS UNIX System Services Command Reference*.

Using the whoami command

The whoami command displays a username associated with the effective user ID, unlike the who am i command which displays the login name.

For example, if you login as 'user1' but then you use the su command to change to 'user2':

For more information about the whoami command, see the <u>whoami</u> command description in <u>z/OS UNIX</u> System Services Command Reference.

Running a TSO/E command

To run a TSO/E command from the shell or in a shell script, simply preface the TSO/E command with either the tso or tsocmd shell command.

Using the tso command

To run a TSO/E command from the shell or in a shell script, you can preface the TSO/E command with the tso shell command; for example:

tso -t tso_command

There are two options you can use:

• Specify the -*t* option to run a command through the TSO/E service routine. The command output is written to stdout. If you specify a relative path name, the command looks for the file in your current directory.

Restriction: TSO/E has some restrictions on the type of commands that can be run using the TSO/E service routine (mini-TSO environment). In summary, you cannot run the following commands in this environment:

- Commands that run authorized
- FIB (foreground initiated background) commands
- Other commands that require the TSO/E task structure; for example, interactive commands such as oedit, where interactive means that the user can interact with the command processing while issuing additional terminal input (subcommands, function keys). For example, once the oedit command is entered, the user can enter more subcommands to add more lines and then quit or exit the command.

For a full description of the restrictions, see the information on IKJTSOEV in <u>z/OS TSO/E Programming</u> Guide. • Specify the -*o* option to run a TSO command as if it had been entered on the OMVS command line and run using the TSO subcommand or function key. If you use a relative path name, the command looks for the file in the working directory of your TSO/E session, which is typically your home directory.

If no option is specified, the following rules are applied in this order:

- 1. If stdout is not a tty, the TSO service routine is used since it is possible that the command output is redirected to a file or piped to another command. Otherwise,
- 2. If the controlling tty supports 3270 pass-through mode, OMVS is used. Otherwise,

3. The TSO service routine is used.

See <u>"Understanding standard input, standard output, and standard error" on page 62</u> for more information about stdin, stdout, and stderr.

The tso command supports several environment variables. For more information about the tso command and the environment variables that are associated with it, see z/OS UNIX System Services Command Reference.

Using the tsocmd command

You may also use the tsocmd shell command to run a TSO/E command from the shell or in a shell script.

Unlike the tso shell command, the tsocmd shell command can be used to issue authorized TSO commands. For more information about the tsocmd shell command and the environment variables associated with it, see *z/OS UNIX System Services Command Reference*.

Using the man command to get online help

Use the man command to get help information about a shell command. The man page is displayed in your shell session, and you can work in the shell while viewing the command. The man syntax is:

man command_name

- To scroll the information in a man page, press <Enter>.
- To end the display of a man page, type q and press <Enter>.

To search for a particular string in a system that has a list of one-line command descriptions, use the -*k* option:

man -k string

For example, to produce a list of all the shell commands for editing, you could type:

man -k edit

You can use the man command to view descriptions of TSO/E commands. To do this, you must prefix all commands with tso.

To view a description of the MOUNT command, enter:

man tsomount

You can also use the man command to view descriptions of dbx subcommands. To do this, you must prefix all subcommands with dbx. For example, to view a description of the dbx alias subcommand, enter:

man dbxalias

For complete information about the man command, see <u>man - Display sections of the online reference</u> manual in *z/OS UNIX System Services Command Reference*.

Shell messages

Messages issued by the z/OS shell and utilities are prefixed with the letters FSUM. The shell messages are documented in *z/OS UNIX System Services Messages and Codes*.

Chapter 7. Working with tcsh shell commands

The shell is, above all, a *programmer's* interface. As a result, the shell commands are strongly slanted towards the needs of a programmer. The tcsh shell has many *general* tools that can help any programmer, and is specifically designed to have syntax similar to the C programming language. In addition, there are a number of commands designed especially for the C programmer.

Specifying shell command options

Most of the commands discussed in this topic accept options. Shell command options are usually specified by a minus sign (–) followed by a single character. For example, the 1s 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 <u>"Listing directory</u> contents" on page 195 for an example.)

ls -1

lists directory names in a single column.

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

ls -1 dir1

to list the contents of dir1 in a single column.

Command options and arguments must be typed as single-byte characters. Additionally, delimiters such as a slash, braces, and parentheses must be typed as single-byte characters.

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

ls dir1 -F

1s lists the contents of **dir1** and then tries to list the contents of the directory, or attributes of the file, called **-F**.

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

ls -- -F

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

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

cat -t -v file

or you could combine the two options into:

cat -tv file

The order of the options is not important:

cat -vt file

is equivalent to the previous version of the command.

Specifying options with accompanying arguments

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

For example, if you read the <u>ps</u> command description in <u>z/OS UNIX System Services Command Reference</u>, you will see that ps accepts an argument of the form:

-u userlist

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

ps -u 'macneil,wellie1'

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

Help for shell command usage

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

For online help information about a command, see <u>"Using the man command to get online help" on page</u> 82.

Understanding standard input, standard output, and standard error

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

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

2. It writes to its standard output file.

- If you invoke a shell command from the shell, a C program, or a REXX program invoked from TSO READY, standard output is directed to your terminal screen by default.
- If you invoke a shell command, REXX program, or C program from the ISPF shell, standard output cannot be directed to your terminal screen. You can specify a z/OS UNIX file or use the default, a temporary file.
- 3. It writes error messages to its *standard error* file.
 - If you invoke a shell command from the shell or from a C program or from a REXX program invoked from TSO READY, standard error is directed to your terminal screen by default.
 - If you invoke a shell command, REXX program, or C program from the ISPF shell, standard error cannot be directed to your terminal screen. You can specify a z/OS UNIX file or use the default, a temporary file.

If the standard output or standard error file contains any data when the command completes, the file is displayed for you to browse.
Using the shell: In the shell, the names for these files are:

- stdin for the standard input file.
- **stdout** for the *standard output* file.
- stderr for the standard error file.

Using TSO/E: When you are invoking the BPXBATCH utility, you can specify these standard files in MVS DD statements, TSO/E ALLOCATE commands, or DYNALLOC macros using the ddnames:

- STDIN for standard input
- STDOUT for standard output
- STDERR for standard error

For more information about BPXBATCH, see "The BPXBATCH utility" on page 145.

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

Redirecting command output to a file

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

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

>filename

For example:

cat file1 file2 file3 >outfile

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

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

>>filename

instead.

Another example:

(sort -u file1 >output) >&outerr

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

Suppose you entered:

sort -u filea >output

In this command, you see two redirections:

• Error output from the sort is redirected to standard output, the display screen.

• The result of the sort is redirected to the file named **output**.

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

```
c89 -o hello -V hello.c >&hello.list
```

Redirecting input from a file

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

mailx JAYD <lessons</pre>

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

Redirecting error output to a file

You can redirect error output from the workstation screen to a file. For example:

(sort -u filea >dev/tty) >& outerr

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

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

(sort -u filea >/dev/tty) >& /dev/null

Dumping nontext files to standard output

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

od file

dumps a file in octal.

od -h file

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

Setting up an alias for a command

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

alias name "string"

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

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

alias renam "mv"

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

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

c file

to get the effect of:

cat file

Defining an alias

If you will be using an alias frequently, put the alias command in your profile file (**\$HOME/.tcshrc**). That way, you do not have to type them in every time you start using the shell. See <u>"Understanding the startup</u> files" on page 49 for more information about customizing your startup files.

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

alias

and the shell displays them.

Arguments in aliases

Any arguments that follow an alias are treated just as if they had been following the command that the alias stands for. For example, if you define the alias \pm as follows:

alias f "ls"

the shell replaces f with 1s, which is the command to list files in a directory.

You can refer to arguments in an alias by simply adding them at the end of the alias as you would with a command. For example:

f -la

would perform the 1s command with the arguments 1a, which will list all the files in the directory in a long directory listing format. And,

f /bin

will list the contents of the /bin directory.

Redefining an alias for a session

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

alias name "string"

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

Setting up an alias for a particular version of a command

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

grep hello file

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

grep -i hello file

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

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

alias grep "grep -i"

From this point on, if you use the command:

grep string file

it is automatically converted to:

grep -i string file

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

As another example, the rm command to delete (remove) a file has an -i option that prompts you to confirm the deletion. The 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 shell checks specially for an alias of the same name, and does the correct thing.

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

Turning off an alias

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

• Get rid of the alias entirely. The command:

unalias rm

gets rid of the rm alias for the session. After this, when you enter rm, you get the real rm command.

• Escape the alias. If you put a backslash in front of an alias, the shell uses the real command rather than the alias. For example:

\rm file

• Specify the full pathname. For example:

/bin/rm file

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

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

Combining commands

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

• You can run a series of commands, one after the other:

Using a semicolon (;) Using && and ||

• You can run more than one command concurrently:

Using a pipe (|) or a filter with a pipe

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

Using a semicolon (;)

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

cd mydir ; ls

Also, if you have defined the alias:

alias l "ls -l"

you can enter:

cd mydir ; l

because you can use aliases such as 1 after a semicolon.

Using && and ||

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

&&

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

Ш

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

Using a pipe

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

command | command | ...

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

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

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

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

ls –l | grep "Jan"

uses 1s to obtain information about 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 that are currently active in the system and pipes the output to grep, which searches for every instance of the string cc. The output from grep is then piped to wc, which counts every line in which the string cc occurs and sends the number of lines to standard output.

Using substitution in commands

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

`command

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

As an example of how a programmer could use command substitution, consider a file called **srclist**, containing the following list of source code 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 1s 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 1s information only for files that were last changed in November. (To be perfectly accurate, it also displays information about files that have the string Nov in their names, too.)

Another useful find option has the form:

find path -ctime number

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

ls -l `find dir -ctime 1`

displays 1s information about 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 z/OS UNIX find command automatically prints its results without – print. However, if you have an existing shell script or compatibility with UNIX systems is important to you, you can use –print.

For more information about the find command, see the find command description in <u>z/OS UNIX System</u> Services Command Reference.

Characters that have special meaning to the shell

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

Characters used with commands

Character

Usage

Pipes the output from one command to a second command; separates commands in a *pipeline*.

Ш

L

Separates two commands. If the command preceding || fails, it runs the following command (Boolean OR operator).

>

Redirects stdout.

<

Redirects stdin.

&

Runs a command in the background, if placed at the end of a command line.

>&

Used for redirecting stdout and stderr.

&&

Separates two commands. If the command preceding && succeeds, it runs the following command (Boolean AND operator).

;

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

()

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

(s_p) is also used to group mathematical operations.

{}

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

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

#

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

\$

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

١

In general, the backslash character turns off the special meaning of the character that follows it. For more information, see <u>"Using a special character without its special meaning" on page 95</u>.

..

A pair of single quotation marks turns off the special meaning of all characters within the quotation marks. For more information, see <u>"Using a special character without its special meaning" on page</u> 95.

....

A pair of double quotation marks turns off the special meaning of the characters within the quotation marks, except that !event, \$var, and `cmd` will show history, variable, and command substitution. See <u>"Using a special character without its special meaning" on page 95 for more information.</u>

Characters used in file names

Character

Usage

/

Separates the components of a file's pathname.

~

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

~valerie/.tcshrc

refers to user VALERIE's .tcshrc file.

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

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

?

••

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

*

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

Redirecting input and output

Character	Usage	Example
<	Redirects input to a specified file.	"Redirecting input from a file" on page 88.
>	Redirects output to a specified file.	"Redirecting command output to a file" on page 87.
>>	Redirects output to be appended to the end of the specified file.	"Redirecting command output to a file" on page 87.
>&	Redirects stdout and stderr.	"Redirecting error output to a file" on page 88.
< <text <i="" encounters="" input="" it="" reads="" standard="" until="">text.</text>	Reads standard input until it encounters <i>text</i> .	This is used in what is called a "here-document." Input is usually typed on the screen or in a shell script. For example, this script creates a file called hello.c , compiles it into hello , and then executes it:
	<pre># create program cat > hello.c << EOF main() { puts("Hello, World!\n"); } EOF # compile program c89 -o hello hello.c #execute program hello</pre>	
		When you run the shell script, it runs the cat $>$

Using a special character without its special meaning

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

\ ...

The backslash

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

echo it\'s me

hello.c command using the input between the

two End_of_File strings.

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.

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 that !event, \$var, and `cmd` will show history, variable, and command substitution.

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*. (In an MVS context, the wildcard character is referred to as a *global character*, or *pattern-matching character*.) A wildcard character is a special character that may be used to save typing in file names in shell commands. The tcsh shell recognizes several different wildcard characters:

```
*
?
```

[]

The * character

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

ls aa*

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

The command:

mv *.c dir1/dir2

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

You can use the * wildcard character in directory names as well as in 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 pathname, the question mark ? can stand for any single character, except a leading dot. For example:

file.?

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

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

ls *.?

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

The square brackets

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

[bch]at

matches bat, cat, or hat.

ls [abc]*

lists all files in the working directory the names of which start with a, b, or c, followed by any other sequence of zero or more characters. In other words, it lists all files whose names start with a, b, or c.

You can specify ranges of characters inside the square brackets by specifying the first character in the sequence, a hyphen (-), and the last character. For example:

[a-m]

This matches any character from a through m.

Suppose, for example, that you want to copy the contents of the working directory into two separate directories. You might enter:

cp [a-m]* dira

to copy all files with names beginning with the letters a through m to the directory dira, and then issue the second command:

cp [n-z]* dirb

to copy the rest of the files to the directory dirb. A command such as:

rm *.[a-z]

removes every file with a suffix consisting of a single lowercase letter.

If the first character inside a bracket construct is an exclamation mark !, the construct matches any character that is not inside the brackets. For example:

ls [!a-m]*

lists any file that does not begin with one of the letters in the range a through m.

In the same way:

rm [!0-9]*

removes any file with a name that does not start with a digit.

Retrieving previously entered commands

In the tcsh shell, you can retrieve previously issued commands using:

- The history command, combined with the ! command
- The two retrieve function keys that are part of the TSO/E OMVS command interface to the shell
- Command-line editing, when you are using an asynchronous terminal interface

Retrieving commands from the history file

The shell records each command that you enter in a file under your *home directory*. This file is called the *history file*; its name is **.history**. If you enter the command:

history

the shell displays the current contents of your history file. Each command is numbered.

You can rerun any of the commands in your history file by typing !, followed by a space, followed by the number of the command you want to use.

For example, suppose that you are a programmer and you enter a complicated command to compile part of a program. The program contains a syntax error, so you call a text editor to edit the source code and correct the problem. Now you want to run the same compile command on the corrected program. You may save yourself a good deal of typing by using:

history

to find out the number of the previous compile command and then running the command with !. For example, if the history file shows you that the command you want to run is number 44, you would type:

! 44

to run the previous compile command.

Another time-saver is to specify your shell prompt as:

set prompt="\!>

in your **.tcshrc** file. The shell prompt is then preceded by the number assigned to the command in the command history file.

If you type ! followed by a space, followed by a string of characters (not beginning with a digit), the shell checks backward through the history file and runs the most recent command that begins with the given string. For instance, look at the compilation example. Suppose you are using the c++ command to compile your program. Then:

! c++

looks back through the history and runs the most recent c++ command. You do not even have to check on the number of the command you want to enter. The shell displays the selected command in the output area of the screen and then runs it.

This *backward-search* feature of ! can search for aliases as well as normal commands. ! searches for the beginning of the command line as you typed it, not the way that the line looked after the alias was replaced.

If you enter !! without a number after it, the shell repeats the most recent command.

Editing commands from the history file

Suppose that you have a sequence of source files named **file1.c**, **file2.c**, **file3.c**, and so on that you want to compile with similar c89 commands. This situation is a little different from the one discussed in the previous topic. You do not want to rerun the *same* command for each file; the command has the same form each time, but you have to specify in a new file name each time.

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

^old_string^new_string

runs a previous *command* but replaces the first occurrence of the *old* string with the *new* string. For example, suppose you compile **file1.c** with:

c89 options file1.c

Then the command:

^file1^file2

tells the shell to look at the previous command and change **file1** to **file2**. The shell makes this change, and then displays and runs the modified command.

^file2^file3

performs the same kind of operation, changing **file2** in the previous command to **file3** and then going ahead with the compilation. This saves you the trouble of retyping all the options for the command.

Using the retrieve function keys

If you are using the OMVS interface, there are two function key settings for retrieving commands:

Retrieve

This key performs a "backward retrieve" function. It retrieves a saved command from a stack of saved input lines, starting with the most recent and moving down to the oldest available line.

FwdRetr

This key is used with the Retrieve key to retrieve commands from the stack of saved input lines. If you press the Retrieve key one too many times and go past the line you want, you can press the FwdRetr key to display the line that was previously retrieved by the Retrieve key.

Press the Retrieve key repeatedly until the command you want to use is displayed on the command line. Once the command is displayed, you can modify the command or use it as it is displayed. Press <Enter> to run the command.

Command-line editing

When you use rlogin or telnet to login to the shell, you can use command-line editing. Command-line editing lets you access commands from your history file, edit them, and run the result. You have already seen this process before, when reading about some of the features of the ! command.

Command editing is useful at those times when you are running the same sequence of commands, or slight variations on the same sequence of commands. The point of command editing is to save yourself the trouble of typing the same thing over and over again—look especially for long commands that normally require a lot of typing. Command editing is also useful when you have made a mistake in typing a command line and wish to correct it.

Using the vi command editor

If you run the command:

bindkey -v

it tells the shell that you want the ability to edit commands the way that you normally edit text with vi; you are set up for vi command editing. Whenever the shell prompts you for input, it is as if the shell puts you into vi insert mode on a new line at the end of the history file. You can type in a new command just as you normally would.

You can also press <Esc> to enter a vi-like command mode. When you enter command mode, you can use the usual cursor movement commands to move around on the command line, or to move up and down in the history file. For example:

- Press the k key to move back to the previous line in the history file (the last command line you entered). Press the k key again, and you move to the line before that.
- Press j and you move forward in the history file.

In this way it is simple to retrieve recent commands from the history file. You can then edit them using standard vi commands. For example, you can use \$ to move to the end of the line, and A to begin appending text to the end of the line. When you have edited the line to produce the command that you want to run, simply press <Enter> to run that line.

As you might expect, you can use these search commands:

/string ?string

to search backwards and forwards through the history file. You can edit the command line with these vi commands:

w

Move to next word

b

Move to previous word

d

delete

С

change

а

append

i

insert

u

undo

and many of the other vi commands. For a complete list of available commands, see the <u>tcsh</u> command description in *z/OS UNIX System Services Command Reference*.

Using the emacs command editor

To set up for emacs command editing, enter:

bindkey -e

This lets you use commands identical to emacs commands to edit your shell command line. For more information, see the <u>tcsh</u> command description in <u>z/OS UNIX System Services Command Reference</u>.

Using file name completion

Tip: File name completion requires the use of the TAB key. This key must be mapped correctly for the feature to work. Most connections through **telnet** and **rlogin** will transmit the TAB information correctly. If you are connected in any other manner, this feature may not work correctly.

The tcsh shell provides a time saving feature for completing file names. Rather than having to type out the entire string to access a file or execute a program, you can type just the first letter or letters and let the shell help you with the rest.

For example, if you have a file called *phonebook*, and you want to list the contents of this file on the screen with the **more** command, you can do so by typing the command, the first letter or letters of the file, and then pressing the TAB key. For example, if you type:

more ph

and then press the TAB key, the shell will provide you with:

more phonebook

you can then press ENTER and execute the command.

If you have more than one file name that matches the letter or letters you have typed, the shell will alert you with a beep. For example, if you have three files, called *list1*, *list2*, and *list3*, and you type:

more li

and press TAB, the beep will sound, and the shell will complete the file name as far as it can:

more list

you must then type 1, 2, or 3 and press ENTER.

If you are unsure of how many files there are, or which one you want, you can type <CRTL-D> when the shell beeps, and you will be provided with matching names. For example:

```
> more list
list1 list2 list3
> more list
```

Underneath the matching names the command prompt is displayed again. Now you can enter the number that you wish and then press ENTER.

If there are no matches for the letter or letters you have typed, the shell will beep, but when you press <CRTL-D>, nothing will be displayed.

You can also use file name completion to aid in changing between directories with long paths. If you keep files in the directory *stuff/data/graphics*, it is easier to use file name completion to access the directory than to type the entire path by hand. For example, if you are in your home directory, and *stuff* is a subdirectory containing *data/graphics*, and you want to change into that directory, you can do the following:

```
cd s [TAB]
cd stuff/.
cd stuff/d [TAB]
cd stuff/data
cd stuff/data/g [TAB]
cd stuff/data/graphics
```

then press ENTER, and the directory change command will execute.

You can find more information about file name completion in *z/OS UNIX System Services Command Reference*.

Using record-keeping commands

Record-keeping commands can be very helpful for programmers. For example, suppose you have a program that is split into several source files. For the sake of simplicity, assume that the source files all have the extension **.c** and are all stored in a subdirectory called **src**. (To read about extensions, see "Naming files" on page 199.)

It is often the case that you want to find out which source files in the subdirectory refer to a particular variable or function. You can do this very simply with the command:

grep 'name' src/*.c

The command checks all the appropriate files in the subdirectory **src** and displays the lines that contain the given *name*. Each line is labeled with the name of the file that contains the line. You can quickly find the use of a function or data object in source files.

As another example of using record-keeping commands, suppose that you are working on a large program and every few days you back up the source code for the program by copying it to a directory in a different file system (as a precaution). You would like to compare the current versions of your source files with one of the saved versions, to find out what changes have been made between the two. The command:

```
diff oldfile newfile
```

prints out all the differences between two versions of a file, making comparisons possible.

The cksum command gives a checksum for each file. If applied to two versions of what was at one time the same file, cksum gives a convenient way to tell if the files are still the same. It does not, however, indicate what the differences are.

The find command also has applications to programming. For example, suppose you are looking for a particular C source program but cannot remember where it is stored.

```
find / -name '*.c'
```

searches all the files and file systems, starting at the root, and displays the names of all files with the **.c** extension.

Finding elements in a file and presenting them in a specific format

awk is a powerful command that can perform many different operations on files. The general purpose of awk is to read the contents of one or more files, obtain selected pieces of information from the files, and present the information in a specified format.

One simple way to use awk is with a command line with the form:

awk '/regexp/ {action}' file

This asks awk to obtain information from the specified file. awk obtains the information by performing the specified *action* on every line in the file that contains a string matching the given regular expression, *regexp*. (For further information, see <u>Regular expressions (regexp)</u> in <u>z/OS UNIX System Services</u> Command Reference.) For example:

awk '/abc/ {print}' file

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

For more discussion on using awk, see Appendix B, "Using awk," on page 281.

Timing programs

The time command lets you time programs to find out how much processor time they actually require. You might use this to compare two versions of a program to see if one runs faster than the other. You can run a program with:

time command-line

where *command-line* is a command line that invokes the program you want to time. time runs the program and displays:

- The total time the program took to execute, labeled real
- The total time spent in the user program, labeled user
- The central processor time spent performing system services for the user, labeled sys

For more information, see the time command description in *z/OS UNIX System Services Command Reference*.

Using the passwd command

You can change a user's password or password phrase with the passwd command:

passwd [-u userid]

The passwd command changes the login password or password phrase for the user ID specified. If *userid* is omitted, the login name associated with the current terminal is used. You are prompted for the new password or password phrase.

For example:

passwd

changes the password or password phrase for the invoker. The invoker is prompted for the old password or password phrase and then for the new value.

Non-superusers can change the password or password phrase for another user if they know the user ID and the current password or password phrase. This example changes the password or password phrase for user ID **bonnie**:

passwd -u bonnie

For more information about the passwd command, see the password command description in <u>z/OS UNIX</u> <u>System Services Command Reference</u>. For information on setting up RACF to enable password phrase support, see z/OS Security Server RACF Security Administrator's Guide.

Switching to superuser or another ID

With the su command, you can switch to any user ID, including the superuser. A user can switch to superuser authority (with an effective UID of 0), if the user is permitted to the BPX.SUPERUSER resource in the FACILITY class within the Resource Access Control Facility (RACF). Either the ISPF shell or the su shell command can be used for switching to superuser authority.

If you do not specify a user ID, the su command changes your authorization to that of the superuser. If you specify a user ID, su changes your authorization to that of the specified user ID.

When you switch to superuser (UID 0) without specifying a user ID, you keep your MVS identity (TSO/E ID). You keep your access authority to MVS data sets, while gaining authority to access any z/OS UNIX files.

When you change user ID by specifying a user ID and password, you assume the MVS identity of the new user ID, even if the user ID has UID 0.

If you use the –s option on the su command, you will not be prompted for a password. Use this option if you have access to the BPX.SRV.*userid* SURROGAT class profile. The *userid* is the MVS user ID associated with the target UID.

To return to your own user ID, type:

exit

This returns you to the shell in which you entered the su command.

For more information, see the su command description in *z/OS UNIX System Services Command Reference*.

Using the whoami command

The whoami command displays a username associated with the effective user ID, unlike the who am i command, which displays the login name.

For example, if you login as user1 and then use the su command to change to 'user2':

command	returned
who am I	user1
whoami	user2

For more information about the whoami command, see the <u>whoami</u> command description in <u>z/OS UNIX</u> System Services Command Reference.

Running a TSO/E command

To run a TSO/E command from the shell or in a shell script, simply preface the TSO/E command with either the tso or tsocmd shell command.

Using the tso command

To run a TSO/E command from the shell or in a shell script, you may preface the TSO/E command with the tso shell command; for example:

tso -t tso_command

There are two options you can use:

• Specify the -*t* option to run a command through the TSO/E service routine. The command output is written to **stdout**. If you specify a relative pathname, the command looks for the file in your current directory.

Restrictions: TSO/E has some restrictions on the type of commands that can be run using the TSO/E service routine (mini-TSO environment). In summary, you cannot run the following commands in this environment:

- Commands that run authorized
- FIB (foreground initiated background) commands
- Other commands that require the TSO/E task structure, i.e., interactive commands such as oedit, where interactive means that the user can interact with the command processing while issuing additional terminal input (subcommands, function keys). For example, once the oedit command is entered, the user can enter additional subcommands to add more lines and then quit or exit the command.

For a full description of the restrictions, see the information on IKJTSOEV in *z/OS TSO/E Programming Guide*.

• Specify the -*o* option to run a TSO command as if it had been entered on the OMVS command line and run using the TSO subcommand or function key. If you use a relative pathname, the command looks for the file in the working directory of your TSO/E session, which is typically your home directory.

If no option is specified, the following rules are applied in this order:

- 1. If **stdout** is not a tty, the TSO service routine is used since it is possible that the command output is redirected to a file or piped to another command. Otherwise,
- 2. If the controlling tty supports 3270 passthrough mode, OMVS is used. Otherwise,
- 3. The TSO service routine is used.

See "Understanding standard input, standard output, and standard error" on page 62 for more information about stdin, stdout, and stderr.

The tso command supports several environment variables. For more information about the tso command and the environment variables associated with it, see z/OS UNIX System Services Command Reference.

Using the tsocmd command

You can also use the tsocmd shell command to run a TSO/E command from the shell or in a shell script.

Unlike the tso shell command, the tsocmd shell command can be used to issue authorized TSO commands. For more information about the tsocmd shell command and the environment variables associated with it, see *z/OS UNIX System Services Command Reference*.

Online help

Two help facilities are available with the shell:

• The man command, which displays help information about a shell command. The man page is displayed in your shell session, and you can work in the shell while viewing the help information.

Using the man command

You can use the man command to get help information about a shell command. The man syntax is:

man command_name

- To scroll the information in a man page, press <Enter>.
- To end the display of a man page, type q and press <Enter>.

To search for a particular string in a system that has a list of one-line command descriptions, use the -k option:

man -k string

For example, to produce a list of all the shell commands for editing, you could type:

man -k edit

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

man tsomount

You can also use the man command to view descriptions of dbx subcommands. To do this, you must prefix all subcommands with dbx. For example, to view a description of the dbx alias subcommand, you would enter:

man dbxalias

For complete information about the man command, see <u>man - Display sections of the online reference</u> manual in *z/OS UNIX System Services Command Reference*.

Shell messages

Messages issued by the tcsh shell and utilities are prefixed with the letters FSUC. See <u>z/OS UNIX System</u> Services Messages and Codes.

[Programming Interface Information] Chapter 8. Writing z/OS shell scripts

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 can have to go through the same sequence of shell commands each week to update the books and produce a report.

To simplify such jobs, the shell lets you run a sequence of commands that have been stored in a text file. For example, the programmer could store all the appropriate compiling and linking commands in a file. A file containing commands in this way is called a *shell script*. After such a file is completed and it is made executable, the programmer can run all the commands in the file by entering the 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 once. Then you can run all the commands in the script by entering the name of the file as a single shell command. A shell script can save you a lot of time and effort if you are working with many files, or if some command lines have several options.
- Reduces the number of errors. If you are typing in ten commands, you have ten chances to make a mistake. With a shell script, however, you can take your time, edit the file carefully and be sure that it is correct before you try to run it.
- Makes it easy for other people to do what you do. For example, consider the bookkeeper example. When the bookkeeper goes on vacation, someone else has to fill in. It is much easier for the substitute bookkeeper to type a single command that does everything correctly than to try to type in the full sequence of commands.

For all these reasons, you will probably find that the use of shell scripts makes your work easier and more productive. This topic provides only a brief overview, but it should give you an idea of how to write and use shell scripts.

[End Programming Interface Information]

Running a shell script

You can run a shell script by typing the name of the file that contains the script. For example, suppose you have a script named **totals.scp** that has three shell commands in it. If you enter:

totals.scp

the shell runs the three commands.

Before you can run a shell script, you must have read and execute permission to the file. Use the chmod and umask commands to set the permissions. See the discussion of permissions in <u>Chapter 18, "Handling</u> security for your files," on page 217.

For another example, suppose you want to compile a collection of files written in the C programming language. You could use the c89, cc, or c++ command. The c89 command, for example, compiles any file **file.c**, link-edits the object module, and produces an executable file. The shell script:

 compiles and link-edits the files and produces an executable file, **outfile**. Notice that in a shell script you precede a comment with a *#*.

If you store this script in an executable file named **compile**, it could be run with the single command compile. A new process is created for the script to run in.

To run a shell script in your current environment, without creating a new process, use the . (dot) command. You could run the **compile** shell script this way:

. compile

If you want to use a shell script that updates a variable in the current environment, run it with the . command.

Tip: You can improve shell script performance by setting the _BPX_SPAWN_SCRIPT environment variable to a value of YES. See "Improving the performance of shell scripts" on page 40 for more information.

Using the magic number

When a script file starts with #!, the kernel's spawn and exec services recognize the file name after the #! as the program to be run. For example, the z/OS UNIX file **/u/userid/util1** contains the following in the start of the file:

#! /u/userid/othershell

The kernel recognizes the magic number (#!) and runs /u/userid/othershell.

Using TSO/E commands in shell scripts

A shell script can include TSO/E commands as well as shell commands, and it can process TSO/E command output. You use the tso shell command to run the TSO/E command. For a discussion of the tso command, see "Using the tso command" on page 81.

Using variables

You can think of shell scripts as *programs* made up of shell commands. To allow more versatile shell scripts, the shell supports many of the features of normal programming languages.

In a conventional programming language, a *variable* is a name that has an associated value. When you want to use the value, you can use the name instead.

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

Creating a variable

The shell also lets you create variables. A shell variable name can consist of uppercase or lowercase letters, plus digits and the underscore character _. The name can have any length, but the first character cannot be a digit. Uppercase letters are distinguished from lowercase ones, so **NAME**, **name**, and **Name** are all *different* names.

To create a shell variable, just enter:

name='string'

as a command to the shell. No spaces are allowed around the =. For example:

HOME='/usr/macneil'

sets up a variable with the name HOME and the value /usr/macneil.

After you set a variable, you refer to it by prefixing its name with a dollar sign (\$). Any command can use the value of a variable by referring to it this way. For example, if **HOME** is set to **/usr/macneil**:

cd \$HOME

is equivalent to:

cd /usr/macneil

Similarly:

cp \$HOME/* /newdir

is equivalent to:

cp /usr/macneil/* /newdir

To change the value of an existing variable, you use a command with the same form as the existing variable. For example:

HOME='/usr/benjk'

changes the value of HOME from /usr/macneil to /usr/benjk.

If the value on the right-hand side of the = sign does not contain spaces, tab characters, or other special characters, you can leave out the single 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 normal variable assignment assigns a *string* to a variable. Thus j gets the string 1+1.

To evaluate an arithmetic expression, you can enter:

```
let "variable=expression"
```

This command line assigns the value of an expression to the given variable. For example:

```
i=1
let "j=$i+1"
echo $j
```

Here j is assigned the value of the expression and the echo command displays the value 2.

You can also use let to change the value of a variable. If you enter:

```
i=1
let "i=$i+1"
echo $i
```

the let command *changes* the value of i. The new value of i is the old value plus 1.

A let command can have any of the standard arithmetic expressions:

-A

Negative A

A*B

A times B

A/B

A divided by B

A%B

Remainder of A divided by B

A+B

A plus B

A-B

A minus B

The standard mathematical order of operations is used, as shown in the way that operations are grouped:

• All unary minus operations are carried out;

• Then any *, /, or % operations (from left to right in the order they appear);

• Then any additions or subtractions (from left to right in the order they appear).

Many operators use special shell characters, so you usually need to put double 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, we have talked about defining shell variables and then using them in later command lines. You can also define a shell variable and then call a shell script that makes use of that variable. But you have to do a certain amount of preparation first.

A shell script is run like a separate shell session. By default, it does not share any variables with your current shell session. If you define a variable **VAR** in the current session, it is *local* to the current session; any shell script that you call will not know about **VAR**.

To deal with this situation, you can export the command; enter:

export VAR

The export command says that you want the variable **VAR** passed on to all the commands and shell scripts that you execute in this session. After you do this, **VAR** becomes *global* and the variable is known to all the commands and shell scripts that you use.

As an example, suppose you enter the commands:

MYNAME="Robin Hood" export MYNAME

Now all your commands can use the **MYNAME** variable to obtain the associated name. You may, for example, have shell scripts that write form letters that contain your name, Robin Hood, obtained from the **MYNAME** variable.

Note: You could use single or double quotation marks to enclose the variable value. See <u>"Quoting variable values" on page 36</u> for more information.

When a script begins running, it automatically inherits all the variables currently being exported. However, if the script changes the value of one of those variables, that change is not reflected to the calling shell—unless you run the script with the dot (.) utility.

By default, any variables created within a shell script are *local* to that script. This means that when another program is run, those variables do not apply in its environment. However, the script can use the export command to turn local variables into global ones. Inside a shell script:

export name

indicates that the variable with the given *name* should be exported. When other programs are run from that script, they inherit the value of all exported variables. However, when the script ends, all its exported variables are lost to the calling shell.

Some variables are automatically marked for export by the software that creates them. For example, if you invoke the shell, the initialization procedure automatically marks the **HOME** variables for export so that other commands and shell scripts can use it. In <u>Chapter 4</u>, "Customizing the z/OS shell," on page 35, you saw that in a typical **.profile** file for an individual user, the **PATH** variable is exported. Exporting **PATH** ensures that search rules and changes to search rules are automatically shared by all shell sessions and scripts.

You must export other variables explicitly, using the export command.

Associating attributes with variables

The typeset command lets you associate attributes with shell variables. This process is analogous to declaring the type of a variable in a conventional programming language. For example:

typeset -i8 y

says that y is an octal integer. In this way, you can make sure that arithmetic with y is always performed in base 8 rather than the usual base 10.

Other attributes may specify how the variable's value is displayed when the variable is expanded. Attributes of this kind are:

-Ln

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.

-RZn

The value should always be displayed with *n* characters, right-justified and with enough leading zeros to fill out the rest of the space.

-Zn

The same as -RZn.

–LZn

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 -1 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 marks it read-only so that the value cannot be changed.

Displaying currently defined variables

The command typeset without any arguments displays the currently defined variables and their attributes. The variation:

typeset -x

displays all the variables currently defined for export.

Using positional parameters — the \$N construct

The sample shell script discussed previously in this topic compiled and link-edited a program stored in a collection of source modules. This information discusses a shell script that can compile and link-edit a C program stored in any file.

To create such a script, you need to be familiar with the idea of *positional parameters*. When the shell encounters a \$N construct formed by a \$ followed by a single digit, it replaces the construct with a value taken from the command line that started the shell script.

- \$1 refers to the first string after the name of the script file on the command line
- \$2 refers to the second string, and so on.

As a simple example, consider a shell script named **echoit** consisting only of the command:

echo \$1

Suppose we run the command:

echoit hello

The shell reads the shell script from **echoit** and tries to run the command it contains. When the shell sees the \$1 construct in the echo command, it goes back to the command line and obtains the first string following the name of the shell script on the command line. The shell replaces the \$1 with this string, so the echo command becomes:

echo hello

The shell then runs this command.

A construct like \$1 is called a *positional parameter*. Parameters in a shell script are replaced with strings from the command line when the script is run. The strings on the command line are called *positional parameter values* or *command-line arguments*.

If you enter:

echoit Hello there

the string Hello is considered parameter value \$1 and the string there is \$2. Of course, the shell script is only:

echo \$1

so the echo command displays only the Hello.

Positional parameters that include a blank can be enclosed in 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 about anyone in the file named Smith.

Using quotation marks to enclose a 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 that the file search contains:

```
grep "$1" *
```

If you enter the command:

search 'two words'

the parameter value 'two words' replaces the construct \$1 in the grep command:

grep "two words" *

If the grep command does not contain the double quotation marks, the parameter replacement would result 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.

Using parameter and variable expansion

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 braces; for example, 7 and {15} are both valid representations of positional parameters. For example, if the command:

echo \$1

appeared in a shell script, it would echo the first positional parameter.

Similarly, a \$ followed by the name of a shell variable (such as **\$HOME**) stands for the value of the variable.

These constructs are called *parameter expansions*. In this sense, the term *parameter* can mean either a positional parameter or a shell variable.

The z/OS shell also supports more complicated forms of parameter expansions, letting you obtain only part of a parameter value or a modified form of the value.

Parameter expansion	Usage
\${parameter:-value}	You can use $fearameter: -value$ in any input to the shell. If <i>parameter</i> currently has a value and the value is not null (for example, a string without characters), the foregoing construct stands for the parameter's value. If the value of the parameter is null, the construct is replaced with the <i>value</i> shown in the brace brackets. For example, a shell script might contain:
	SHELL=\${SHELL:-/bin/sh}
	If the SHELL variable currently has a value, this simply assigns SHELL its own current value. However, if the value of SHELL is null, the given assignment will have the value of /bin/sh . The value after : – can be thought of as a <i>backup</i> 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 $ferrer:=value$ is similar to the previous form; the difference is that if the given <i>parameter</i> does not currently have a value, the given <i>value</i> is assigned to <i>parameter</i> , and then the new value of parameter is used. Thus the := form actually assigns a value if the <i>parameter</i> does not already have one. In this case, <i>parameter</i> must be a variable; it cannot be a positional parameter.

Parameter expansion	Usage
\${parameter:?message}	The expansion \${parameter:?message} is related to the previous two forms. If the value of the given parameter is null, the given message is displayed. If the construct is being used inside a shell script, the script ends with an error status. For example, you might have:
	<pre>cp \$1 \${2:?"Must specify a directory name"}</pre>
	In this case, the message following the ? is displayed if there is no second positional parameter. If you omit the <i>message</i> , the shell prints a standard message. For example, you could just enter:
	cp \$1 \${2:?}
	to get the standard error message.
\${parameter:+replacement}	The construct \${parameter:+replacement} might be thought of as the opposite of the preceding expansions. If parameter has not been assigned a value, or has a null value, this construct is just the null string. If parameter does have a value, the value is ignored and the replacement value is used in its place. Thus, if a shell script contains:
	echo \${1:+"There was a parameter"}
	the echo command displays:
	There was a parameter
	if the script was invoked with a parameter. If no parameter was specified, the echo command has nothing to echo.
\${parameter#pattern}	The construct \${parameter#pattern} is evaluated by expanding the value of parameter and then deleting the smallest leftmost part of the expansion that matches the given pattern of pathname wildcard characters. For example, suppose that the variable NAME stands for a file name. You might use:
	\${NAME#*/}
	to remove the highest-level directory from the pathname. If:
	NAME="user/dir/subdir/file.c"
	then:
	\${NAME#*/}
	expands to:
	dir/subdir/file.c

Parameter expansion	Usage
\${parameter##pattern}	The construct \${parameter##pattern} removes the largest leftmost part that matches the pattern. For example, if:
	NAME="user/dir/subdir/file.c"
	then:
	\${NAME##*/}
	yields:
	file.c
	The wildcard character * stands for any sequence of characters. In this situation, it stands for everything up to the final slash.
\${parameter%pattern}	The construct \${parameter%pattern} removes the smallest rightmost part of the parameter expansion that matches pattern. Thus if:
	NAME="user/dir/subdir/file.c"
	then:
	\${NAME%.?}
	stands for:
	user/dir/subdir/file
\${parameter%%pattern}	Similarly, \${parameter%%pattern} stands for the expansion of parameter without the longest rightmost string that matches pattern. Using the previous example of NAME,
	\${NAME%%/*}
	stands for:
	user

Using special parameters in commands and shell scripts

The z/OS shell has a variety of special parameters that may be used in command lines and shell scripts.

Parameter	Expands to
\$@	The complete list of positional parameters, each separated by a single space. If \$@ is quoted, the separate arguments are each quoted; for example:
	echo "\$@"
	is equivalent to:
	"\$1" "\$2" "\$3"
	If the positional parameters are all file names:
	cp \$@ dir
	copies all the files to the given directory dir .
\$*	The complete list of positional parameters. If \$* is quoted, the result is concatenated into a single argument, with parameters separated by the first character of the value of the shell variable IFS . For example, if the first character of IFS is a comma, then:
	echo "\$*"
	displays the parameters with separating commas:
	"\$1,\$2,\$3"
\$#	The number of positional parameters passed to this shell script. This number can be changed by several shell commands (for example, set or shift); see <u>z/OS</u> <u>UNIX System Services Command Reference</u> .
\$?	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 the test command 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.

Examining the nature of a file

Table 4 on page 118 lists the test commands that you can use to determine the nature of a file.

Table 4: Using the test command to examine the nature of a file.	
Syntax of the test command Questions asked	
test -d pathname	Is <i>pathname</i> a directory?
test -f pathname	Is pathname a file?
test -r pathname	Is <i>pathname</i> readable?
test -w pathname	Is <i>pathname</i> writable?

Comparing the age of two files

Table 5 on page 118 lists the test commands that you can use to compare the age of two files.

Table 5: Using the test command to compare the age of two files.	
Syntax of the test command Questions asked	
test <i>file1</i> -ot <i>file2</i>	Is <i>file1</i> older than <i>file2</i> ?
test <i>file1</i> -nt <i>file2</i>	Is <i>file1</i> newer than <i>file2</i> ?

Comparing the values of two numbers

Table 6 on page 118 lists the test commands that you can use to compare the values of two numbers.

Table 6: Using the test command to compare the values of two numbers.	
Syntax of the test command	Questions asked
testA-eqB	Is A equal to B?
testA-neB	Is A not equal to B?
testA-gtB	Is A greater than B?
testA-ltB	Is A less than B?
testA-geB	Is A greater than or equal to B?
testA-leB	Is A less than or equal to B?

Comparing two strings

Table 7 on page 118 lists the test commands that you can use to compare two strings.

Table 7: Using the test command to compare two strings.	
Syntax of the test command Questions asked	
test str1 = str2	Is <i>str1</i> equal to <i>str2</i> ?
test str1 != str2	Is str1 not equal to str2?

Testing whether strings are empty

Table 8 on page 118 lists the test commands that you can use to determine whether strings are empty.

Table 8: Using the test command to test whether strings are empty.	
Syntax of the test command Questions asked	
test -z string	Is string empty?
test -n string	Is string not empty?

Any of these tests will also work if you put square brackets ([]) around the condition instead of using the test command. For example, test 1 -eq 1 is the equivalent of [1 -eq 1].

The double square bracket [[test_expr]] syntax is also supported. The double square bracket ([[]]) also supports additional tests over the **test** command, and there are some subtle differences between the tests (for example, string equal vs. pattern matching).

The result of test is either true or false. 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 1s command to display the contents of the directory.

Any number of commands may come between the then and the fi that ends the control structure. For example, you might have written:

```
if
   test -d $1
then
   echo "$1 is a directory"
   ls $1
fi
```

This example also shows that the commands do not have to begin on the same line as then, and the condition being tested does not have to begin on the same line as if. The condition and the commands are indented to make them stand out more clearly. This is a good way to make your shell scripts easier to read.

Another form of the if conditional is:

```
if condition
then commands
else commands
fi
```

If the condition is true, the commands after the then are run; otherwise, the commands after the else are run. For example, suppose you know that the string associated with the variable *pathname* is the name of either a directory or a file. Then you could write:

```
if
   test -d $pathname
then
   echo "$pathname is a directory"
   ls $pathname
else
   echo "$pathname is a file"
    cat $pathname
fi
```

If the value of *pathname* is the name of a file, this shell script uses echo to display an appropriate message, and then uses cat to display the contents of the file.

The final form of the if control structure is:

```
if condition1
then commands1
elif condition2
then commands2
elif condition3
then commands3
...
else commands
fi
```

elif is short for "else if" In this example, if *condition1* is true, *commands1* are run; otherwise, the shell goes on to check *condition2*. If that is true, *commands2* are run; otherwise, the shell goes on to check *condition3* and so on. If none of the test conditions are true, the *commands* after the else are run. Here is an example of how this can be used:

```
if test ! "$1"
then
    echo "no positional parameters"
elif test -d $1
then
    echo "$1 is a directory"
    ls $1
elif test -f $1
then
    echo "$1 is a file"
    cat $1
else
    echo "$1 is just a string"
fi
```

The test after the if determines if the value of the first positional parameter, \$1, is an empty string. If so, there are no positional parameters, and 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 1s (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 the *commands*. The shell then goes back to check the *condition*. If it is still true, the shell runs the *commands* again, and so on, until the *condition* is found to be false.

As an example of how this can be used, suppose you want to run a program named **prog** 100 times to get an idea of the program's average running speed. The following shell script does the job:

```
i=100
date
while test $i -gt 0
do
prog
let i=$i-1
done
date
```

The script begins by setting a variable i to 100. It then uses the date command to get the current date and time.

Next the script runs a while loop. The test condition says that the loop should keep on going as long as the value of *i* is greater than zero. The commands of the loop run **prog** and then subtract 1 from the *i* variable. In this way, *i* goes down by 1 each time through the loop, until it is no longer greater than 0. At this point, the loop stops and the final instruction of the script prints out the date and time at the end of the loop. The difference between the starting time and the ending time should give some idea of how long it took to run the program 100 times.

(Of course, the shell itself takes some time to perform the test and to do the calculations with *i*. If **prog** takes a long time to run, the time spent by the shell is relatively unimportant; if **prog** is a quick program, the extra time that the shell takes may be large enough to make the timing incorrect.)

You can rewrite this shell script to make it a little more efficient:

```
i=100
date
while let "(i=$i-1) >= 0"
do
    prog
done
date
```

In this example, the let command is the condition of the while loop. It gives *i* a new value and then compares this value to zero. The advantage of this way of writing the program is that it does not have to call test to make the comparison; this speeds up the loop and makes the time more accurate.

The for loop

The final control structure to be examined is the for loop. It has the form:

```
for name in list
do commands
done
```

The parameter *name* should be a variable name; if this variable doesn't exist, it is created. The parameter *list* is a list of strings separated by spaces. The shell begins by assigning the first string in *list* to the variable *name*. It then runs the *commands* once. Then the shell assigns the next string in *list* to *name*, and repeats the *commands*. The shell runs the *commands* once for each string in *list*.

As a simple example of a shell script that uses for, consider:

```
for file in *.c
do
c89 $file
done
```

When the shell looks at the for line, it expands the expression *****. c to produce a *list* containing the names of all files (in the working directory) that have the suffix **.c**. The variable *file* is assigned each of the names in this list, in turn. The result of the for loop is to use the c89 command to compile all **.c** files in the working directory. You could also write:

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:

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 it 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:

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.

For more information about the test command, see <u>z/OS UNIX System Services Command Reference</u>. The section that discusses reserved words in <u>z/OS UNIX System Services Command Reference</u> contains information about the [[...]] form.

Using functions

A *shell function* is similar to a function in C: It is a sequence of commands that do a single job. Typically, a function is used for an operation that you tend to do frequently in a shell script. Before you can call a function in a shell script, you must define it in the script. After the function is defined, you can call it as many times as you want in the script.
As an example, consider the following piece of a shell script, showing the function definition and how the function is called in the shell script:

```
function td
ş
   if test -d "$1"
                                   # test if first argument is directory
   then
        curdir=$(pwd)
                                   # set curdir to working directory
                                   # change to specified directory
        cd $1
                                   # run specified command
        $2
       cd $curdir
                                   # change back to working directory
                                   # return 0 if successful
       return 0
   else
       echo $1 "is not a directory"
                                   # return 1 if not successful
       return 1
   fi
td /u/turbo/src.c ls
                                   # invoking the function
```

The purpose of td is to go to a specified directory, run a single command, and then return to the directory from which the function was called.

To run a function, specify the function's name followed by whatever arguments it expects. To run the function td, specify the function name followed by a directory name and a command name, as shown in the last line of the foregoing example.

As you see in the td example, a function can also return a value. If the statement:

```
return expression
```

appears inside a function, the function ends and the value of *expression* is returned as the status, or *result*, of the function. In general, the returned value:

- 0 means that the function has succeeded in its task.
- 1 means that the function has failed.

Anytime you need to repeatedly perform the same sequence of commands in a shell script, consider defining a function to do the sequence of commands. This lets you organize a large script into smaller blocks of subroutines.

In order to make a shell function available as a shell command, the function definition must be processed by the shell that will execute the command. Typically, the user sets up a shell script (such as **\$HOME/.setup**) that contains all of the function definitions, and sets the ENV variable to the pathname of that shell script. As the number of functions in this script grows, the time to process the function definitions causes shell initialization time to increase.

Autoloading functions

Autoloading improves the performance of shell initialization by delaying function definition processing until the first use. Functions that are not used by a particular user are never read by the shell, thus avoiding the processing of unused functions. The FPATH variable allows flexibility in accessing directories with system-wide, group, or personal function definitions.

FPATH is defined with the same format as the PATH variable. FPATH is a list of directories separated by colons. These directories contain only function definitions and should not contain the current working directory.

To use autoloading, place frequently used and shared functions in a directory pointed to by the FPATH variable and specify the function name on an autoload or typeset -f command in the user's ENV setup script.

The autoload command identifies functions that are not yet defined. The first time that an autoload function is called within the shell, the shell searches FPATH directories for a file with the same name as the function definition. If a matching file with the same name as the function is found, it is processed and stored in the shell's memory for subsequent execution. The matching file contains the function definition for the autoload function. Other function definitions may be found in this matching file, and if so, they

will be defined to the shell when the file is processed. For information about how to set up the FPATH search path, see <u>"Customizing the FPATH search path: The FPATH variable" on page 40</u>.

[Programming Interface Information] Chapter 9. Writing tcsh shell scripts

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

- A programmer may always use the same commands to compile source code, and link the resulting object code.
- A bookkeeper may have to go through the same sequence of shell commands each week to update the books and produce a report.

To simplify such jobs, the shell lets you run a sequence of commands that have been stored in a text file. For example, the programmer could store all the appropriate compiling and linking commands in a file. A file containing commands in this way is called a *shell script*. After such a file is completed and it is made "executable," the programmer can run all the commands in the file by entering the 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 once. Then you can run all the commands in the script by entering the name of the file as a single shell command. A shell script can save you a lot of time and effort if you are working with many files, or if some command lines have several options.
- Reduces the number of errors. If you are typing in ten commands, you have ten chances to make a mistake. With a shell script, however, you can take your time, edit the file carefully, and get it right before you try to run it.
- Makes it easy for other people to do what you do. For example, consider the bookkeeper example. When the bookkeeper goes on vacation, someone else has to fill in. It is much easier for the substitute bookkeeper to type a single command that does everything correctly than to try to type in the full sequence of commands.

For all these reasons, you will probably find that the use of shell scripts makes your work easier and more productive. This topic provides only a brief overview, but it should give you an idea of how to write and use shell scripts.

[End Programming Interface Information]

Running a shell script

You can run a shell script by typing the name of the file that contains the script. For example, suppose you have a script named **totals.scp** that has three shell commands in it. If you enter:

totals.scp

the shell runs the three commands.

Before you can run a shell script, you must have read and execute permission to the file. Use the chmod and umask commands to set the permissions. See the discussion of permissions in <u>Chapter 18</u>, "Handling <u>security for your files," on page 217</u>. See the descriptions of chmod and umask in <u>z/OS UNIX System</u> <u>Services Command Reference</u>.

For another example, suppose you want to compile a collection of files written in the C programming language. You could use the c89, cc, or c++ command. The c89 command, for example, compiles any file **file.c**, link-edits the object module, and produces an executable file. The shell script:

c89 -c file1.c file2.c
c89 -o outfile file1.o file2.o file3.c

compile only
outfile for executable

compiles and link-edits the files and produces an executable file, **outfile**. Notice that in a shell script you precede a comment with a *#*.

If you store this script in an executable file named **compile**, it could be run with the single command compile. A new process is created for the script to run in.

To run a shell script in your current environment, without creating a new process, use the source command. You could run the **calculate** shell script this way:

source calculate

Should you want to use a shell script that updates a variable in the current environment, run it with the source command.

Tip: To improve shell script performance, set the _BPX_SPAWN_SCRIPT environment variable to NO when using the tcsh shell. This variable is intended only for use with the z/OS shell. If this variable is inherited from a z/OS shell session, put

#!/bin/tcsh

as the first line in your tcsh shell scripts to avoid any errors. If tcsh is your login shell, you should unset _BPX_SPAWN_SCRIPT, because it is only used for increasing performance of z/OS shell scripts.

Using the magic number

All tcsh scripts must have # as the first character of the script. When a script file starts with #!, the kernel's spawn and exec services recognize the file name after the #! as the program to be run. It is recommended that the first line of all tcsh scripts look like:

#!/bin/tcsh

with **/bin/tcsh** being the location of tcsh on the z/OS system. The kernel recognizes the magic value (#!) and runs **/bin/tcsh**.

Using TSO/E commands in shell scripts

A shell script can include TSO/E commands as well as shell commands, and it can process TSO/E command output. You use the tso shell command to run the TSO/E command. For a discussion of the tso command, see "Using the tso command" on page 81.

Using variables

You can think of shell scripts as programs that are made up of shell commands. To allow more versatile shell scripts, the shell supports many of the features of normal programming languages.

In a conventional programming language, a variable is a name that has an associated value. When you want to use the value, you can use the name instead.

Creating a shell variable

The shell also lets you create variables. A shell variable name can consist of uppercase or lowercase letters, plus digits and the underscore character _. The name can have any length, but the first character

cannot be a digit. Uppercase letters are distinguished from lowercase ones, so **NAME**, **name**, and **Name** are all *different* names.

To create a shell variable, just enter:

set name='string'

as a command to the shell. For example:

set home='/usr/adams'

sets up a variable with the name **home** and the value **/usr/adams**.

After you set a variable, you refer to it by prefixing its name with a dollar sign (\$). Any command can use the value of a variable by referring to it this way. For example, if **home** is set to **/usr/adams**:

cd \$home is equivalent to: cd /usr/adams Similarly: cp \$home/* /newdir

is equivalent to:

cp /usr/adams/* /newdir

To change the value of an existing variable, you use a command with the same form as the existing variable. For example:

set home='/usr/benjk'

changes the value of home from /usr/adams to /usr/benjk.

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

```
set i=1
set j=$i+1
echo $j
```

The output of echo is 1+1, because a normal variable assignment assigns a *string* to a variable. Thus j gets the string 1+1.

To evaluate an arithmetic expression, you can enter:

@ variable=expression

This command line assigns the value of an expression to the given variable. For example:

```
i=1
@ j=$i + 1
echo $j
```

Here j is assigned the value of the expression and the echo command displays the value 2.

You can also use @ to change the value of a variable. If you enter:

i=1 @ i=\$i + 1 echo \$i

the @ command *changes* the value of i. The new value of i is the old value plus 1.

An @ command can have any of the standard arithmetic expressions:

-A

Negative A

A * B

A times B

A / B A divided by B

А % В

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:

@ 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:

@i = ((5 + 2) * 3)

assigns 21 to i.

Note: @ does not work with numbers that have fractional parts. It works only with integers.

Setting environment variables

Up to this point, we have talked about defining shell variables and then using them in later command lines. You can also define a shell variable and then call a shell script that makes use of that variable. But you have to do a certain amount of preparation first.

A shell script is run as a child process to the parent shell. By default, the child process does not share any variables with the parent. If you define a variable **var** in the parent shell, it is *local* to the current session; any shell script, or child process, that you call will not inherit **var**.

To deal with this situation, you can enter the following:

setenv var [value]

The setenv command says that you want the variable **var** passed on to all the child processes that you execute in this session. After you do this, **var** becomes inherited and the variable is known to all the commands and shell scripts that you use.

As an example, suppose you enter the commands:

setenv myname "Friar Tuck"

Now all your child processes can use the **myname** variable to obtain the associated name. You may, for example, have shell scripts that write form letters that contain your name, Friar Tuck, obtained from the **myname** variable.

Note: You could use single or double quotation marks to enclose the variable value. See <u>"Quoting variable</u> values" on page 50 for more information.

When a script or child process begins running, it automatically inherits all the environment variables passed on to it. However, if the script changes the value of one of those variables, that change is *not* passed back to the parent process —unless you run the script with the source utility.

By default, any variables created within a shell script are *local* to that script. This means that when another program is run, those variables do not apply in its environment. However, the script can use the setenv command to turn shell variables into global environment ones. Inside a shell script:

setenv name [value]

indicates that the variable with the given *name* should be defined as an environment variable. When other programs are run from that script, they inherit the value of all environment variables. However, when the script ends, all its environment variables are lost to the calling shell.

Some variables are automatically inherited by the software that creates them. For example, if you invoke the shell, the initialization procedure automatically marks the **HOME** variables for environment variables so that other commands and shell scripts can use it. In <u>Chapter 5</u>, "Customizing the tcsh shell," on page <u>49</u>, you saw that in a typical **.tcshrc** file for an individual user, the **PATH** variable is an environmental variable. Making the **PATH** variable an environmental variable ensures that search rules and changes to search rules are automatically shared by all shell sessions and scripts.

Using positional parameters — the \$N construct

The sample shell script discussed previously compiled and link-edited a program stored in a collection of source modules. This topic discusses a shell script that can compile and link-edit a C program stored in any file.

To create such a script, you need to be familiar with the idea of *positional parameters*. When the shell encounters a \$N construct formed by a \$ followed by a single digit, it replaces the construct with a value taken from the command line that started the shell script.

- \$1 refers to the first string after the name of the script file on the command line
- \$2 refers to the second string, and so on.

As a simple example, consider a shell script named **echoit** consisting only of these commands:

```
#!/bin/tcsh #
echo $1
```

Suppose we run the command:

echoit hello

The shell reads the shell script from **echoit** and tries to run the command it contains. When the shell sees the \$1 construct in the echo command, it goes back to the command line and obtains the first string following the name of the shell script on the command line. The shell replaces the \$1 with this string, so the echo command becomes:

echo hello

The shell then runs this command.

A construct like \$1 is called a *positional parameter*. Parameters in a shell script are replaced with strings from the command line when the script is run. The strings on the command line are called *positional parameter values* or *command-line arguments*.

If you enter:

echoit Hello there

the string Hello is considered parameter value \$1 and the string there is \$2. Of course, the shell script is only:

echo \$1

so the echo command displays only the Hello.

Positional parameters that include a blank can be enclosed in quotes (single or double). For example:

echoit "Hello there"

echoes the two words instead of just one, because the two words are handled as one parameter.

Returning to a compile and link example, a programmer could write a more general shell script as:

c89 -c \$1.c c89 -o \$1 \$1.o

If this shell script were named **clink**, the command:

clink prog

would compile and link **prog.c**, producing an executable file named **prog** in the working directory. In the same way, the command:

clink dir/prog2

would compile and link dir/prog2.c. The shell script compiles and links a C program stored in a single file.

As another example of a shell script containing a positional parameter, suppose that the file **lookup** contains:

grep \$1 address

where **address** is a file containing names, addresses, and other useful information. The command:

lookup Smith

displays address information on anyone in the file named Smith.

Using quotes to enclose a construct in a shell script

A \$N construct in a shell script can be enclosed in double or single 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 does not contain the double quotation marks, the parameter replacement results 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

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 braces; for example, 7 and {15} are both valid representations of positional parameters. For example, if the command:

echo \$1

appeared in a shell script, it would echo the first positional parameter.

Similarly, a \$ followed by the name of a shell variable (such as **\$HOME**) stands for the value of the variable.

These constructs are called *parameter expansions*. In this sense, the term *parameter* can mean either a positional parameter or a shell variable.

The tcsh shell also supports more complicated forms of parameter expansions, letting you obtain only part of a parameter value or a modified form of the value.

Modifier	Description
r	Root of value
е	Extension of value
h	Head of value
t	Tail of value

For example, to extract only part of a file name, you can add one of these modifiers as follows:

File name	r	е	h	t
/usr/bin/vi.txt	/usr/bin/vi	txt	/usr/bin	vi.txt
/u/bobby/mail	/u/bobby/mail	empty	/u/bobby	mail
storybook.pdf	storybook	pdf	empty	storybook.pdf
INSTALL	INSTALL	empty	empty	INSTALL

Using special parameters in commands and shell scripts

The tcsh shell has a variety of special parameters that can be used in command lines and shell scripts. These parameters are listed in the Variable Substitution topic of the tcsh command description in z/OS UNIX System Services Command Reference.

Using control structures

The shell provides facilities similar to those found in programming languages. It offers these *control structures,* which are related to programming control structures:

- The if conditional
- The while loop
- The for loop

The if conditional

An if conditional runs a sequence of commands if a particular condition is met. It has the form:

```
if (expr) command
```

The end of the commands is indicated by endif. For example, you could have:

```
if ( -d $1 ) then
ls $1
endif
```

This tests to see if the string associated with the first positional parameter, \$1, is the name of a directory. If so, it runs an 1s command to display the contents of the directory.

Any number of commands can come between the then and the endif that ends the control structure. For example, you might have written:

```
if ( -d $1 ) then
    echo "$1 is a directory"
    ls $1
endif
```

This example also shows that the commands do not have to begin on the same line as then, and the condition being tested does not have to begin on the same line as if. The condition and the commands are indented to make them stand out more clearly. This is a good way to make your shell scripts easier to read.

Another form of the if conditional is:

```
if (expr) then
commands
else
commands
endif
```

If the condition is true, the commands after the then are run; otherwise, the commands after the else are run. For example, suppose you know that the string associated with the variable *pathname* is the name of either a directory or a file. Then you could write:

```
if ( -d $pathname ) then
        echo "$pathname is a directory"
        ls $pathname
        else
        echo "$pathname is a file"
        cat $pathname
        endif
```

If the value of *pathname* is the name of a file, this shell script uses echo to display an appropriate message, and then uses cat to display the contents of the file.

The final form of the if control structure is:

```
if (expr1) then
commands1
else if (expr2) then
commands2
else if (expr3) then
```

commands3 else commands endif

In this example, if *expr1* is true, *commands1* are run; otherwise, the shell goes on to check *expr2*. If that is true, *commands2* are run; otherwise, the shell goes on to check *expr3* and so on. If none of the test conditions are true, the *commands* after the else are run. Here is an example of how this can be used:

```
if ( ! $?argv ) then
        echo "no positional parameters"
else if ( -d $1 ) then
        echo "$1 is a directory"
        ls $1
else if ( -f $1 ) then
        echo "$1 is a file"
        cat $1
else
        echo "$1 is just a string"
endif
```

The test after the if determines if the value of the first positional parameter, \$1, is an empty string. If so, there are no positional parameters, and 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 1s (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 (expr)
commands
end
```

The shell first tests to see if *condition* (expr) is true. If it is, the shell runs the *commands*. The shell then goes back to check the *condition*. If it is still true, the shell runs the *commands* again, and so on, until the *condition* is found to be false.

As an example of how this can be used, suppose you want to run a program named **prog** 100 times to get an idea of the program's average running speed. The following shell script does the job:

The script begins by setting a variable i to 100. It then uses the date command to get the current date and time.

Next the script runs a while loop. The condition says that the loop should keep on going as long as the value of *i* is greater than zero. The commands of the loop run **prog** and then subtract 1 from the *i* variable, similar to C programming language syntax. In this way, *i* goes down by 1 each time through the loop, until it is no longer greater than 0. At this point, the loop stops and the final instruction of the script prints out the date and time at the end of the loop. The difference between the starting time and the ending time should give some idea of how long it took to run the program 100 times.

(Of course, the shell itself takes some time to perform the condition and to do the calculations with *i*. If **prog** takes a long time to run, the time spent by the shell is relatively unimportant; if **prog** is a quick program, the extra time that the shell takes may be large enough to make the timing incorrect.)

The foreach loop

The final control structure to be examined is the foreach loop. It has the form:

```
foreach name (wordlist)
commands
end
```

The parameter *name* must be a variable name; if this variable does not exist, it is created. The parameter *list* is a list of strings separated by spaces. The shell begins by assigning the first string in *list* to the variable *name*. It then runs the *commands* once. Then the shell assigns the next string in *list* to *name*, and repeats the *commands*. The shell runs the *commands* once for each string in *list*.

As a simple example of a shell script that uses foreach, consider:

```
foreach file ( *.c )
    c89 $file
end
```

When the shell looks at the foreach line, it expands the expression *****. c to produce a *list* containing the names of all files (in the working directory) that have the suffix **.c**. The variable *file* is assigned each of the names in this list, in turn. The result of the foreach loop is to use the c89 command to compile all **.c** files in the working directory. You could also write:

```
foreach file ( *.c )
    echo $file
    c89 $file
end
```

so that the shell script displayed each file name before compiling it. This would let you keep track of what the script was doing.

As you can see, the foreach loop is a powerful control structure. The *list* can also be created with command substitution, as in:

```
foreach file ( `find . -name "*.c" -print` )
        echo $file
        c89 $file
end
```

Here the find command finds all **.c** files in the working directory, and then compiles these files. This is similar to the previous shell script, but it also looks at subdirectories of the working directory.

Combining control structures

You can combine control structures by nesting (that is, putting one inside another). For example:

```
foreach file ( `find . -name "*.c" -print` )
    if ( -M $file > -M $1 ) then
        echo $file
        c89 -c $file
    endif
end
```

This shell script takes one positional parameter, giving the name of a file. The script looks in the working directory and finds the names of all **.c** files. The if control structure inside the foreach loop tests each file to see if it is older than the file named on the command line. If the **.c** file is older, echo displays the name, and the file is compiled. You can think of this as making a set of files up to date with the file name specified on the command line.

Chapter 10. Using job control in the shells

When you enter a shell command, you start a process, the execution of a function. When you enter that command, the shell runs it in its own process group. As such, it is considered a separate *job* and the shell assigns it a job identifier, which is a small number known only to the shell. (A shell job identifier identifies a shell job, not an MVS job.) When the process completes, the system displays the shell prompt.

The system also assigns a process group identifier (PGID) and a process identifier (PID). When only one command is entered, the PGID is the same as the PID. The PGID can be thought of as a systemwide identifier. If you enter more than one command at a time using a pipe, several processes, each with its own PID, are 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 (PID) is a unique identifier assigned to a process while it runs. When the process ends, its PID is returned to the system. Each time you run a process, it has a different PID (it takes a long time for a PID to be reused by the system). You can use the PID to track the status of a process with the ps command or the jobs command, or to end a process with the kill command.

PGID

Each process in a process group shares a process group ID (PGID), which is the same as the PID of the first process in the process group. This ID is used for signaling related processes.

If a command starts just one process, its PID and PGID are the same.

PPID

A process that creates a new process is called a *parent process*; the new process is called a *child process*. The parent process ID (PPID) becomes associated with the new child process when it is created. The PPID is not used for job control.

Several job control commands can either take as input or return the job identifier, process identifier, or process group identifier: bg, fg, jobs, kill, and wait.

The nice and renice commands can be used to change the priority of processes. Their use is dependent on the way performance groups have been prioritized at your installation; check with your system administrator for information about using nice and renice to change job priority.

Running several jobs at once (foreground and background)

The shell can run more than one job at a time. While one is running in the foreground, one or more can be running in the background.

After you enter a command, you see the output from the command displayed on your screen. You cannot enter any other commands until the shell prompt (\$ or >) appears. This command has run as a *foreground job*. Commands that take a few seconds to complete are convenient to run in the foreground.

You may prefer to run as *background jobs* those shell commands that take longer to run, because they prevent you from running any other commands while they are running in the foreground. The shell does *not* wait for the completion of a background command before returning a prompt to you. Instead, while the command runs in the background, you can continue entering other commands on the command line.

In TSO/E, a *background job* is one that is typically entered at a workstation by a SUBMIT command. Like a TSO/E background job or a batch job, a z/OS UNIX *background job* runs without user interaction.

You can use any of these methods to run a shell background job:

- Start the job in the background when you first enter it.
- Move a job from the foreground to the background.

• Use JCL with BPXBATCH. This utility is discussed in "The BPXBATCH utility" on page 145.

Starting a job in the background with an ampersand (&)

To start a command as a background job, end the command line with an ampersand (&). For example:

sort myfile >myout &

When the background job starts to run, the system:

- 1. Assigns it a job identifier, a process group ID (PGID), and a process ID (PID).
- 2. Displays the job identifier (in brackets) and one or more PIDs (more than one if there is a pipe).
- 3. 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.

Tip: Note the PID numbers and the job number when you create a background job; you can use them to check the status of the job or to end it.

Unlike a batch job, a shell job running in the background directs its output to standard output, your workstation screen. If you do not want to have this output interfering with your work in the foreground, remember to redirect the output to a file when you start a background command. After the output is redirected, you can look at it whenever it is convenient.

A background job can be suspended. A background job that attempts to read from **stdin** is suspended until it is made the foreground process. Therefore, if a program reads from **stdin**, you may want to redirect **stdin** from a file. Also, if the tostop setting of the terminal is enabled (you can set or query this by using the stty command), output from a background job causes the job to be suspended.

Moving a job to the background

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

- 1. Stop the job by entering <EscChar-Z>. A message displays the job identifier.
- 2. Enter the bg command. You may need to specify the job identifier with bg if there is more than one stopped job. If you do not specify a job identifier, bg uses the most recently stopped job.

A message displays the job identifier and the command that is running in the background.

Moving a job to the foreground

When you want to move a job from the background to the foreground, use the fg command. If there are multiple background jobs, you need to supply the job identifier preceded by a % sign. For example:

fg %7

Setting up job tracing

The bpxtrace command provides details on job activity. It enables you to start a job with tracing activated, or enables you to dynamically activate tracing for a job that is already running. For example, to activate job tracing for the echo command, enter:

```
bpxtrace -c -f format "echo hello"
```

This command produces tracing output (to stdout) and formats the output in one line per trace record format. For more information about the bpxtrace command, see <u>z/OS UNIX System Services Command</u> Reference.

Checking the status of jobs

You can use 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 that are currently running, based on the job identifier; it also reports on the status of stopped processes and completed processes. If you use the -1 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 that is being run. In this case the status message shown in the z/OS shell is:

[1] 720902 + Stopped (SIGTSTP) myprog|grep write 720902 alive -sh 458759 alive -sh

In this case:

- The job identifier is 1 (from [1]).
- The PIDs of the processes are 720902 and 458759.
- The PGID is 720902 (the PID of the first process in the process group).

The status message for the tcsh shell is similar to that in the previous example.

Using the ps command

You can use the ps command to display a list of your processes that are currently running and obtain additional information about those processes. (Only a superuser or a user with appropriate permissions can obtain information about all processes.)

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

PID	TTY	TIME	COMMAND
262148	ttyp0000	2:46	/bin/sh
196614	ttyp0000	0:22	./myprog
65543	ttyp0000	0:13	/bin/grep
196616	ttvp0000	2:07	/bin/ps

PID

This is a PID displayed as a decimal value.

TTY

The name of the controlling terminal, if any. The *controlling terminal* is the workstation that started the process. On a system with more than one workstation, the names of the workstations that have started processes are listed here.

TIME

The amount of central processor time the process has used since it began running.

COMMAND

The name of the command or program that started the process. The display indicates which directory the command or program is found in. For example, the ps command is in **/bin**.

Usually, just issuing ps will tell you all you need to know about your current processes. However, there are a number of options you can use to tailor the displayed information. For example, you can use the –a option to display only processes associated with a terminal, not the system processes. Read the <u>ps</u> command description in *z/OS UNIX System Services Command Reference*.

Canceling a job

Often you will start a job and then decide to interrupt it before it completes. You can do this regardless of whether the job is running in the foreground or background.

Canceling a foreground job

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

Canceling a background job

To cancel a background job, use the kill command. To be able to kill a process, you must own it. (The superuser, however, can kill any process except init.)

Before you can cancel a background job, you need to know either a PID, job identifier, or PGID. You can use the jobs command to determine any of these.

The format of the kill command in the z/OS shell is:

kill [-s signal name] [pid] [job-identifier]

The format of the kill command in the tcsh shell is:

kill [-signal name] [pid] [job-identifier]

To kill one process, use its PID.

Example: To kill a process with the PID 717, issue:

kill 717

Any other processes in the job-from a pipe-would not be killed.

To kill a particular process group, you can use a job identifier or a negative PGID.

• You can use the job identifier for one process in the group preceded with a % to kill every process in the group. In the z/OS shell, use:

kill -s KILL %7

In the tcsh shell, use:

kill -KILL %7

• You can use a negative PGID to kill every process in a process group. (The PGID is the PID for the first process in the process group.) For example, in the z/OS shell:

kill -s KILL -- -123456

will kill every process in the process group with PGID 123456.

In the tcsh shell:

kill -KILL -123456

will kill every process in the process group with PGID 123456.

Stopping and resuming a job

Occasionally, you might 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 %.

Examples::

1. In the z/OS shell, to stop a background job with the job identifier 3, issue:

kill -s -STOP %3

2. In the tcsh shell, to stop a background job with the job identifier 3, issue:

kill -STOP %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, to have Time for tea display on your screen when the command whose job identifier is 7 finishes running, issue:

wait %7; print "Time for tea"

Exiting the shell with background jobs running

When you exit the shell, any stopped background jobs are terminated. But if you have a background job in the running state, you can exit the shell without terminating it.

In the z/OS shell, the default setting set -m runs background jobs in a separate process group. Jobs in a separate process group are not sent a **SIGHUP** signal when you exit the shell. With the default -m setting, background jobs continue to run after you exit the shell.

In the tcsh shell, use **NOHUP** to exit the shell with background jobs running.

For the OMVS interface:

To exit with a background job running, use the quit subcommand. (Type quit and press the Subcommand function key or switch to subcommand mode and enter the quit command.) A background job that is running will continue running.

If you are using the OMVS interface and you use the exit command to exit the shell while you have a shell background job running, OMVS may send this message:

The shell process ended, but the session did not end automatically. You may need to run the QUIT subcommand to end the session.

For the Asynchronous terminal interface:

To exit when a background job is running, type <Ctrl-D> or use the exit command. A background job that is running will continue running. You do not get any indication that a background job is running.

Changing the default in the z/OS shell

If you change the setting to +m, background jobs end when you exit the shell. If you have changed the setting to +m and you want to start a long-running command and have it continue running after you exit the shell, use the nohup command and an ampersand (&):

```
nohup 'command-line' &
```

For example:

nohup sort -u file1 >output 2>>outerr &

Ending the nohup command with an & makes the command run in the background, even after you exit the shell.

Comparison of shell background jobs and MVS batch jobs

Table 9 on page 140 compares two methods for submitting a background job:

- Typing an & (ampersand) after the shell command
- Using JCL with BPXBATCH. This utility is discussed in "The BPXBATCH utility" on page 145.

Table 9: Comparison of running a background job from the shell and from MVS		
Topic Shell (command &)		JCL with BPXBATCH
Starting the job	Background jobs start running immediately.	Background jobs are put in a queue; there may be a wait until the job starts running.
Interactive access	You can see output from the job displayed on the terminal. You can move the job to	Background jobs run separately; you cannot interact with them.
	and then move it to the background again.	However, if you redirect output to a file in the file system, from your interactive shell session you could periodically browse the output file to see what is in it. You could do this with any of these commands: cat, pg, more, obrowse, or the TSO/E OBROWSE command.

Table 9: Comparison of running a background job from the shell and from MVS (continued)		
Торіс	Shell (command &) JCL with BPXBATCH	
System limits	Due to system limits on the number of processes per user, multiple background jobs run by the same user could fail at some point.	Due to system limits on the number of processes per user, multiple background jobs run by the same user could fail at some point.
Managing the job	You can use ps, kill, bg, fg and jobs on the background job.	You can use ps and kill on the background job.
Impact on system	Creates an immediate demand on the system to support another address space. This could degrade performance for all users.	The system determines when it is a reasonable time to run batch jobs. Batch work can be suspended during periods of heavy interactive workload.

Chapter 11. Using z/OS UNIX from batch, TSO/E, and ISPF

Note: This information is directed towards users of the z/OS shell. Most examples pertain to the z/OS shell and not the tcsh shell.

You can access z/OS UNIX services from batch, TSO/E, or ISPF, using:

- Job control language (JCL) to run shell scripts or z/OS UNIX application programs as batch (background) jobs. This information describes the JCL that supports the z/OS UNIX file system. For more general information about JCL, see z/OS MVS JCL User's Guide. and z/OS MVS JCL Reference.
- Executable files in batch. An *executable file* is any file that can be run as a program. An executable file can be a load module (which is a member of a PDS), a program object (which is either a member of a PDSE or a file in the z/OS UNIX file system), or an interpreted file (such as a REXX EXEC). For a file to be treated as an executable file, it must have execute permission allowed for the invoker.
- BPXBATCH, a utility that can do the following:
 - Run executable files in batch.
 - Run shell commands and executable files from the TSO/E READY prompt.
- TSO/E commands designed to work with MVS data sets. See the section on using commands to work with directories and files and also the section on copying data between the z/OS UNIX file system and MVS data sets for more information. For the complete command descriptions, see the section on TSO/E commands in z/OS UNIX System Services Command Reference.
- REXX programs that are written using z/OS UNIX extensions called syscall commands.
- The ISPF shell.

JCL support for z/OS UNIX

JCL data definition (DD) statements use a data definition name (ddname) to specify the data to be used by the program that you are submitting as a batch job. The ddname is used in two places:

- 1. In your application program. Here the ddname refers to nonspecific data, rather than a specific data set name or path name.
- 2. In the JCL used to submit the application program as a background job. Here it binds the nonspecific reference in the program to a specific data set name or path name.

You can specify a z/OS UNIX file in the JCL for user-written applications or for IBM-supplied services, such as:

- DFSMS, Program Management Binder, a prelinker, or a linkage editor
- BPXBATCH
- The TSO/E OCOPY command

Note: In this discussion, references to JCL also apply to the equivalent dynamic allocation functions.

The PATH keyword

You can use the PATH keyword on a JCL DD statement to specify the path name for a z/OS UNIX file. When you use the PATH keyword, you can also use these keywords:

• **PATHOPTS** to indicate the access for the file (for example, read or read-write) and to set the status for the file (for example, append, create, or truncate). This is analogous to the option arguments on the C open() function.

Note: If you specify either OCREAT or OCREAT together with OEXCL on the PATHOPTS parameter and the file does not exist, z/OS UNIX performs an open() function. The options from PATHOPTS, the path name from the PATH parameter, and the options on PATHMODE (if specified) are specified in the open().z/OS UNIX uses the close() function to close the file before the application program receives control.

- **PATHMODE** to indicate the permissions, or file access attributes, to be set when a file is being created. This is analogous to the mode arguments of the open() function.
- **PATHDISP** to indicate how MVS should handle the file when the job step ends normally or abnormally. This performs the same function as the DISP parameter for a data set.

If PATHOPTS and PATHMODE are absent from the DD statement, an application needs to supply defaults for the options and mode, or issue an error message and fail.

The DSNTYPE keyword

There are two related subparameters on the DSNTYPE keyword of the DD statement:

- HFS (hierarchical file system)
- PIPE (named pipe)

For more information about the JCL keywords, see *z*/OS *MVS JCL Reference*.

Using the ddname in an application

Instead of using data set names or path names in an application, you can use a ddname; then in the JCL, you associate a specific data set or file with that ddname.

Note: The parent process's allocations, for both data sets and files, are not propagated by fork()and are lost on exec(), except for STEPLIB.

You have a choice of two methods for accessing data sets and files in an application:

- The ANSI C function fopen().
- The OPEN macro.

The fopen() function

The fopen() function recognizes and handles the difference between a ddname associated with a data set (DSN keyword) or with a path name (PATH keyword).

Example: Issue:

fopen("dd:FRED", "r+")

Result: The fopen() function takes the ddname FRED, determines if FRED refers to a ddname for a file or a data set, and opens it. Once a file is opened, fread() and fwrite() can access the data

The OPEN macro

The OPEN macro can open a z/OS UNIX file specified with the PATH keyword or an MVS data set specified with the DSN keyword. The macro supports DD statements that specify the PATH parameter only for data control blocks that specify DSORG=PS (EXCP is not allowed). DFSMSdfp supports BSAM and QSAM interfaces to these types of files:

- Regular files
- Character special files (null files only)
- FIFO special files
- Symbolic links

You cannot open directories or external links.

For more information about BSAM and QSAM interface support for access to z/OS UNIX files, see <u>z/OS</u> DFSMS Macro Instructions for Data Sets.

Specifying a ddname in the JCL

In the JCL for a job, you use a DD statement to associate a ddname with the name of a specific MVS data set or z/OS UNIX file.

To specify a file, use the PATH keyword.

Example: To associate the path name for the file /u/fred/list/wilma with the ddname FRED, specify:

//FRED DD PATH='/u/fred/list/wilma'

At another time, you might specify a different file to be associated with the ddname FRED.

To specify a data set, use the DSN keyword.

Example: To associate the data set FRED.LIST.WILMA with the ddname FRED, specify:

//FRED DD DSN=FRED.LIST.WILMA,DISP=SHR

At another time, you might specify a different data set to be associated with the ddname FRED.

Using the submit command

The submit command submits JCL from the shell. By using this command you do not need to open a TSO session to submit JCL. This command accepts the following as input:

- One or more pathnames
- One or more sequential data set or partitioned data set member names
- Standard input.

For example, to submit a job that resides in the z/OS UNIX file buildjcl.jcl, enter the following:

submit buildjcl.jcl

For more information about the submit command, see z/OS UNIX System Services Command Reference.

The **BPXBATCH** utility

BPXBATCH is a utility that you can use to run shell commands or executable files through the batch facility. You can invoke BPXBATCH from a batch job or from the TSO/E environment (as a command, through a CALL command, or from a CLIST or REXX EXEC).

Note: This document provides some examples of how you can use BPXBATCH. For more detailed information about BPXBATCH, see the description of the BPXBATCH utility and the detailed discussion on using BPXBATCH to run executable files under MVS environments in <u>z/OS UNIX System Services Command</u> *Reference*.

BPXBATCH has logic in it to detect when it is running from a batch job. By default, BPXBATCH sets up the stdin, stdout, and stderr standard streams (files) and then calls the exec callable service to run the requested program. The exec service ends the current job step and creates a new job step to run the target program. Therefore, the target program does not run in the same job step as the BPXBATCH program; it runs in the new job step created by the exec service. In order for BPXBATCH to use the exec service to run the target program, all of the following must be true:

- BPXBATCH is the only program running on the job step task level.
- The _BPX_BATCH_SPAWN=YES environment variable is not specified.
- The STDOUT and STDERR ddnames are not allocated as MVS data sets.

If any of the these conditions is not true, then the target program runs either in the same job step as the BPXBATCH program or in a WLM initiator in the OMVS subsys category. The determination of where to run the target program depends on the environment variable settings specified in the STDENV file and on the attributes of the target program.

Restriction: File and data set allocation considerations vary when a BPXBATCH or BPXBATSL request is processed in the same address space via local spawn or forked to another address space. Allocations for any files and data sets other than **stdin**, **stdout**, **stderr**, or **stdenv** and STEPLIB are not available to a program when BPXBATCH uses fork() or exec (STEPLIB EXCLUDED) to run a program in another address space. Data sets that are allocated in JCL, TSO, or an application may conflict with data sets used by BPXBATCH.

Aliases for BPXBATCH

BPXBATSL, BPXBATA2, and BPXBATA8 are provided as aliases for BPXBATCH that use a local spawn to run in the same address space.

BPXBATSL

BPXBATSL performs a local spawn, but does not require resetting of environment variables. BPXBATSL behaves exactly like BPXBATCH and allows local spawning whether the current environment is set up or not. For more information, see the <u>BPXBATCH</u> command in <u>z/OS UNIX System Services Command</u> *Reference*.

BPXBATA2 and BPXBATA8

BPXBATA2 and BPXBATA8 are provided as APF-authorized alternatives to BPXBATSL. BPXBATA2 and BPXBATA8 provide the capability for a target APF-authorized z/OS UNIX program to run in the same address space as the originating job, allowing it to share the same resources, such as allocations and the job log. See the <u>BPXBATCH</u> utility in *z/OS UNIX System Services Command Reference* for details and restrictions on using these interfaces.

Defining standard input, output, and error streams for BPXBATCH

z/OS XL C/C++ programs require that the standard streams, stdin, stdout, and stderr, be defined as either a file or a terminal. Many C functions use stdin, stdout, and stderr. For example:

- getchar() obtains a character from stdin.
- printf() writes output to stdout.
- perror() writes output to stderr.

(For more information about stdin, stdout, and stderr, see <u>"Understanding standard input, standard</u> output, and standard error" on page 62.)

Guidelines for defining stdin, stdout, and stderr

For BPXBATCH, the default for stdin and stdout is /dev/null.

The default for stderr is the same as what is defined for stdout. For instance, if you define stdout to be /tmp/output1 and you do not define stderr, then both printf() and perror() direct their output to /tmp/output1.

Tip: If you define stdin, it must be a z/OS UNIX file.

If you define stdout or stderr, it can be a z/OS UNIX file or an MVS data set.

If you use an MVS data set for stdout or stderr:

- It can be a sequential data set, a partitioned data set (PDS) member, a partitioned data set extended (PDSE) member, or SYSOUT.
- It must have a nonzero logical record length (LRECL) and a defined record format (RECFM). Otherwise, BPXBATCH will redirect the DD to/dev/null and issue message BPXM081I, indicating the redirection of the effected ddname.

Ways to define stdin, stdout, and stderr

You can define stdin, stdout, and stderr in the following ways:

 With the The TSO/E ALLOCATE command, using the ddnames STDIN, STDOUT, and STDERR. For example, the following command allocates the z/OS UNIX file /u/turbo/myinput to the STDIN ddname:

ALLOCATE DDNAME(STDIN) PATH('/u/turbo/myinput') PATHOPTS(ORDONLY)

 The following command allocates the MVS sequential data set TURBO.MYOUTPUT to the STDOUT ddname:

```
ALLOCATE DDNAME(STDOUT) DSNAME('TURBO.MYOUTPUT') VOLUME(volser) DSORG(PS)
SPACE(10) TRACKS RECFM(F,B) LRECL(512) NEW KEEP
```

• A JCL DD statement, using the ddnames STDIN, STDOUT, and STDERR

The following JCL allocates the z/OS UNIX file /u/turbo/myinput to the STDIN ddname:

//STDIN DD PATH='/u/turbo/myinput',PATHOPTS=(ORDONLY)

The following JCL allocates member M1 of a new PDSE TURBO.MYOUTPUT.LIBRARY to the STDOUT ddname and directs STDERR output to SYSOUT:

```
//STDOUT DD DSNAME=TURBO.MYOUTPUT.LIBRARY(M1),DISP=(NEW,KEEP),DSNTYPE=LIBRARY,
// SPACE=(TRK,(5,1,1)),UNIT=3390,VOL=SER=volser,RECFM=FB,LRECL=80
//STDERR DD SYSOUT=*
```

• Redirection, using <, >, and >>

Even if stdout currently defaults to /dev/null, entering the following command from the TSO/E command prompt redirects the output of the ps -el command to be appended to the file /tmp/ ps.out:

BPXBATCH SH ps -el >>/tmp/ps.out

For more information about defining the standard streams for BPXBATCH, see the detailed discussion on using BPXBATCH in the appendix of *z/OS UNIX System Services Command Reference*.

Passing environment variables to BPXBATCH

When you are using BPXBATCH to run a program, you typically pass the program a file that sets the environment variables. If you do not pass an environment variable file when running a program with BPXBATCH, or if the HOME and LOGNAME variables are not set in the environment variable file, those two variables are set from your logon RACF profile. LOGNAME is set to the user name, and HOME is set to the initial working directory from the RACF profile.

Note: When using BPXBATCH with the SH option (SH is the default), environment variables specified in the STDENV DD are overridden by those specified in /etc/profile and .profile (which overrides /etc/profile). This is because SH causes BPXBATCH to execute a login shell that runs the **/etc/profile** script and runs the user's .profile.

To pass environment variables to BPXBATCH, you define a file containing the variable definitions and allocate it to the STDENV ddname. The file can be one of the following:

- A z/OS UNIX file identified with the ddname STDENV.
- An MVS data set identified with the ddname STDENV.

Guidelines for defining STDENV

The default for STDENV is /dev/null.

The following guidelines apply when you specify a z/OS UNIX file for STDENV:

- It must be a text file defined with read-only access.
- Specify one variable per line, in the format variable=value. Environment variable names must begin in column 1.
- The file cannot have sequence numbers in it.

If you use the ISPF editor to create the file, set the sequence numbers off by typing NUMBER OFF on the command line before you begin typing the data. If sequence numbers already exist, type UNNUM to remove them and then type NUMBER OFF.

The following guidelines apply when you specify an MVS data set for STDENV:

- It must be a sequential data set, a PDS member, a PDSE member, or an JCL in-stream data set.
- The record format can be fixed or variable (unspanned).
- Specify one environment variable per record, in the format variable=value. Environment variable names must begin in column 1. Do not use terminating nulls.
- The data set cannot have sequence numbers in it.

If you use the ISPF editor to create the file, set the sequence numbers off by typing NUMBER OFF on the command line before you begin typing the data. If sequence numbers already exist, type UNNUM to remove them and then type NUMBER OFF.

• Trailing blanks are truncated for in-stream data sets, but not for other data sets.

Ways to define STDENV

You can define the STDENV environment variable file in the following ways:

• The TSO/E ALLOCATE command.

Example: The environment variable definitions reside in the MVS sequential data set TURBO.ENV.FILE.

```
ALLOCATE DDN(STDENV) DSN('TURBO.ENV.FILE') SHR
```

• A JCL DD statement. To identify a z/OS UNIX file, use the PATH operand and specify PATHOPTS=ORDONLY.

Example: The environment variable definitions reside in the z/OS UNIX file u/turbo/env.file.

```
//STDENV DD PATH='u/turbo/env.file',PATHOPTS=ORDONLY
```

• An JCL in-stream data set.

Example: The environment variable definitions immediately follow the STDENV DD statement.

```
//STDENV DD *
variable1=aaaaaaa
variable2=bbbbbbbb =
variable5=ffffffff
/*
```

Trailing blanks are truncated for in-stream data sets, but not for other data sets.

• SVC 99 dynamic allocation, if you are running BPXBATCH from a program.

For more information about defining STDENV, see the detailed discussion about using BPXBATCH in <u>z/OS</u> UNIX System Services Command Reference.

Example: Setting up code page support in a STDENV file

To enable national language support for BPXBATCH, set the locale environment variables to your desired locale in the STDENV file. For example, to use the Danish locale, you could put these lines in the file:

```
LANG=Da_DK.IBM-277
LC_ALL=Da_DK.IBM-277
```

After you allocate this file to STDENV, you can test it by typing:

OSHELL echo \$HOME

The path name of your home directory should be displayed, instead of just \$HOME.

_BPX_BATCH_SPAWN and _BPX_BATCH_UMASK environment variables

BPXBATCH uses two environment variables for execution that are specified by STDENV:

- _BPX_BATCH_UMASK=0755
- _BPX_BATCH_SPAWN=YES|NO

_BPX_BATCH_UMASK allows the user the flexibility of modifying the permission bits on newly created files instead of using the default mask (when PGM is specified).

Valid characters for the mask value are the octal digits 0 to 7, inclusive. If an invalid character is found, that character and all subsequent characters to the right are ignored. For example, 0348 is interpreted as 0034 and 0586 is interpreted as 0005.

Note: This variable is overridden by umask (usually set from within /etc/profile) if BPXBATCH is invoked with the SH option (SH is the default). SH causes BPXBATCH to execute a login shell, which runs the /etc/profile script (and runs the user's .profile) and which might set the umask before execution of the intended program.

_BPX_BATCH_SPAWN causes BPXBATCH to use spawn instead of fork/exec and allows data definitions to be carried over into the spawned process. When _BPX_BATCH_SPAWN is set to YES, spawn will be used. If it is set to NO, which is equivalent to the default behavior, fork/exec will be used to execute the program.

If _BPX_BATCH_SPAWN is set to YES, you must consider two other environment variables that affect spawn (BPX1SPN):

• _BPX_SHAREAS=YES|NO|REUSE

When _BPX_SHAREAS is YES or REUSE, the child process that is created by spawn will run in the same address space as the parent's under these conditions:

- The child process is not setuid or setgid to a value different from the parent.
- The spawned file name is not an external link or a sticky bit file.
- The parent has enough resources to allow the child process to reside in the same address space.
- The NOSHAREAS extended attribute is not set.

When _BPX_SHAREAS is NO, the child and parent are run in separate address spaces.

• _BPX_SPAWN_SCRIPT=YES

When _BPX_SPAWN_SCRIPT is YES, the spawn will treat the specified file as a shell script and will invoke the shell to run the shell script.

Setting _BPX_SPAWN_SCRIPT=YES improves shell script performance. See <u>"Improving the performance</u> of shell scripts" on page 40 for more information. For more information about spawn, see <u>spawn</u> (BPX1SPN, BPX4SPN) — Spawn a process in <u>z/OS UNIX System Services Programming: Assembler Callable</u> Services Reference.

Passing parameter data to BPXBATCH

Normally, you pass parameters to BPXBATCH using the parameter string—either in a batch job by using the PARM= parameter on the JCL EXEC statement (see <u>"Invoking BPXBATCH in a batch job" on page 151</u>) or in TSO by typing them on the command line (see <u>"Invoking BPXBATCH from the TSO/E environment"</u> on page 154). The format of the BPXBATCH parameter string is:

SH|PGM shell_command|shell_script|program_name [arg1...argN]

In a batch job, BPXBATCH only allows up to 100 bytes for the parameter string due to JCL limitations. In a TSO command environment, the maximum length of a parameter string is 32,754 bytes. However, BPXBATCH supports the use of a parameter file to pass much longer parameter data—up to 65,536 (64K) bytes.

To pass parameters to BPXBATCH using a parameter file, you define a file containing the parameter data and allocate it to the ddname STDPARM. The parameter file can be one of the following:

- A z/OS UNIX text file
- An MVS data set

The default is to use the parameter string specified on the TSO command line or in the PARM= parameter of the JCL EXEC statement. If the STDPARM ddname is defined, BPXBATCH uses the data found in the specified file rather than what is found in the parameter string or in the STDIN ddname.

Guidelines for defining STDPARM

The contents of the STDPARM file must follow the same format as the BPXBATCH parameter string.

The following guidelines apply when you specify a z/OS UNIX or an MVS data set for STDPARM:

- A z/OS UNIX file must be a text file that the user has read access to. An MVS data set must be a sequential data set, a PDS member, a PDSE member, or a JCL in-stream data set.
- The record format can be fixed or variable (unspanned).
- For in-stream data sets: with the SH option, trailing blanks are not truncated. Records in in-stream data sets are concatenated with blanks as separator characters, and the string remaining after the SH token is passed as a single argument to a /bin/sh -c command. For the PGM option, the string is divided not only at line boundaries but also at blanks within a line.
- The file or data set should not have sequence numbers in it.

Tip: If you use the ISPF editor to create the file, set the sequence numbers off by typing NUMBER OFF on the command line before you begin typing the data. If sequence numbers already exist, type UNNUM to remove them and then type NUMBER OFF.

Ways to define STDPARM

You can define the STDPARM parameter file by using one of the following:

• The TSO/E ALLOCATE command

Example: The parameter data to be passed to BPXBATCH resides in the MVS sequential data set TURBO.ABC.PARMS.

ALLOCATE DDNAME(STDPARM) DSN('TURBO.ABC.PARMS') SHR

• A JCL DD statement. To identify a z/OS UNIX file, use the PATH operand and specify PATHOPTS=ORDONLY.

Example: The parameter data resides in the z/OS UNIX file /u/turbo/abc.parms.

//STDPARM DD PATH='/u/turbo/abc.parms',PATHOPTS=ORDONLY

Example: The BPXBATCH parameter data resides in member P1 of the MVS PDSE TURBO.PARM.LIBRARY.

//STDPARM DD DSN=TURBO.PARM.LIBRARY(P1),DISP=SHR

• An JCL in-stream data set

The BPXBATCH parameter data immediately follows the STDPARM DD statement. Trailing blanks are truncated for in-stream data sets, but not for other data sets.

Example: The following invokes the echo shell command.

```
//STDPARM DD *
SH echo "Hello, world!"
/*
```

Example: Consider the following shell script called myscript.sh. This shell script writes to stdout the first three arguments that are passed to it.

#!/bin/sh
#Write arguments 1 through 3 to stdout
echo \$1
echo \$2
echo \$3

The following is one way to define STDPARM to run the script:

```
//STDPARM DD *
SH /myscript.sh AAAA BBBB CCCC
/*
```

Here is another way, placing the arguments on separate lines:

```
//STDPARM DD *
SH /myscript.sh AAAA
BBBB
CCCCC
/*
```

Result: Both of these STDPARM definitions produce the following output:

AAAA BBBB CCCC

• SVC 99 dynamic allocation, if you are running BPXBATCH from a program

For more information about defining STDPARM for BPXBATCH, see the detailed discussion about using BPXBATCH *z/OS UNIX System Services Command Reference*.

Invoking BPXBATCH in a batch job

You can create a batch job that invokes BPXBATCH to run a z/OS UNIX shell command or executable file.

The JCL to invoke BPXBATCH looks like this:

```
//jobname JOB ...
//stepname EXEC PGM=BPXBATCH,PARM='SH|PGM program_name [arg1...argN]'
```

where:

- When SH is specified, *program_name* is the name of a shell command or a file containing a shell script. SH is the default; therefore, you can omit the PARM= and use STDIN to define the name of the shell script to be invoked. BPXBATCH invokes the login program to run the shell as a login shell. BPXBATCH always runs the shell found in the user's RACF OMVS Segment.
- When PGM is specified, *program_name* is the name of an executable file that is stored in a z/OS UNIX file. Inadvertent use of a shell script with PGM may result in a process that will not end as expected, and will require use of the kill -9 pid command to force termination.
- You can supply optional arguments, *arg1...argN*, to *program_name*. For SH, the entire string after the SH is passed to the login shell without further parsing. For PGM, the arguments are broken at blanks and passed separately. You cannot use quotes in the parameter to pass arguments that contain blanks.
- You can omit the PARM= and, instead, place the parameter data in a file or data set defined by STDPARM.

In the job, you can supply DD statements to define any of the resources discussed previously using the following ddnames:

DDname

Description

STDIN

Standard input (see "Defining standard input, output, and error streams for BPXBATCH" on page 146)

STDOUT

Standard output (see <u>"Defining standard input, output, and error streams for BPXBATCH" on page</u> 146)

STDERR

Standard error (see "Defining standard input, output, and error streams for BPXBATCH" on page 146)

STDENV

Environment variable definitions (see "Passing environment variables to BPXBATCH" on page 147)

STDPARM

BPXBATCH parameter data (see "Passing parameter data to BPXBATCH" on page 149)

Note:

- 1. If you specify data sets in a STEPLIB DD statement, all the data sets should be cataloged.
- 2. UNIT= and VOL=SER= parameters are not propagated to the process that is being executed by BPXBATCH unless the process is run locally by BPXBATCH via the setting of the _BPX_SHAREAS and _BPX_BATCH_SPAWN environment variables: _BPX_SHAREAS=YES and _BPX_BATCH_SPAWN=YES.
- 3. If the job needs to run with a group other than your default group, you need to code GROUP=grpname on the job card to specify the group your job needs to run under. For BPXBATCH, the group needs to have an OMVS segment and a GID defined for it.
- 4. If your job requires a REGION size greater than the default on your system, you may receive this abend code:

ABEND 4093 reason code 0000001c

To fix this, use a larger REGION size.

Example: The following invokes BPXBATCH with a region size of 8 M:

//SHELLCMD EXEC PGM=BPXBATCH,REGION=8M,PARM='SH shell_command'

Example: Running a shell script in batch

You can use BPXBATCH to run a shell script through batch and redirect the output and error messages to z/OS UNIX files or MVS data sets. Because the default is PARM= 'SH', the PARM= is not specified in the following example. The shell script associated with the STDIN ddname is invoked. You can allocate STDIN, STDOUT, and STDERR as z/OS UNIX files, using the PATH operand on the DD statements. You can also allocate STDOUT and STDERR as MVS data sets.

Example: User TURBO runs a shell script in batch, as follows:

- The STDIN ddname defines a shell script to be invoked, /u/turbo/bin/myscript.sh.
- STDOUT defines a file to which to write the standard output, /u/turbo/bin/mystd.out.
- STDERR defines a file to which to write standard error messages, /u/turbo/bin/mystd.err.

//jobname	JOB	
//stepname	EXEC	PGM=BPXBATCH,REGION=8M
//STDIN	DD	PATH='/u/turbo/bin/myscript.sh',PATHOPTS=(ORDONLY)
//STDOUT	DD	PATH='/u/turbo/bin/mystd.out',PATHOPTS=(OWRONLY,OCREAT),
11		PATHMODE=SIRWXU
//STDERR	DD	PATH='/u/turbo/bin/mvstd.err',PATHOPTS=(OWRONLY,OCREAT),
11		PATHMODE=SIRWXU

Example: The following JCL is similar to the previous example and produces equivalent results but uses the PARM= string to specify the shell script to be run:

//jobname	JOB	
//stepname	EXEC	PGM=BPXBATCH,REGION=8M,
11		PARM='SH /u/turbo/bin/myscript.sh'
//STDOUT	DD	PATH='/u/turbo/bin/mystd.out', PATHOPTS=(OWRONLY, OCREAT),
11		PATHMODE=SIRWXU
//STDERR	DD	PATH='/u/turbo/bin/mystd.err',PATHOPTS=(OWRONLY,OCREAT),
//		PATHMODE=SIRWXU

Example: Running a shell command in batch

In the following example, BPXBATCH runs the shell command compress to compress the file /usr/lib/ junk. To start the next JCL job step before the compress command completes, the parameter string is specified as:

```
SH nohup compress /usr/lib/junk & sleep 1
```

If, instead, the parameter string is specified as:

SH compress /usr/lib/junk

the job step waits for the compress shell command to end. For short-running commands, this is fine.

For long-running commands, however, where you want to use BPXBATCH to start a shell command in the background and not wait for completion, you must specify the parameter string like this:

SH nohup command args & sleep 1

SH starts a login shell to parse and run the command. The login shell parses the &, signifying that the command is to run asynchronously (in the background), and forks a child process to run the nohup command. In the child process, the nohup shell command (which takes another command as an argument) prevents the process from being terminated when the login shell returns to BPXBATCH.

In parallel with the nohup processing, the login shell runs the sleep command. Running the sleep command delays the login shell from returning to BPXBATCH until the child process has had enough time (1 second) to protect itself from being terminated. The login shell returns to BPXBATCH, while the child process continues to run the compress command.

Example: User TURBO runs the compress shell command in batch, as follows:

- STDPARM defines an in-stream data set containing the parameter string.
- STDERR defines a file to which to write error messages, /u/turbo/bin/mystd.err.
- The STDIN and STDOUT files default to /dev/null.
- The STEPLIB is propagated for the execution of the shell and for any processes created by the shell.

//jobname JOB	
//stepname EXE	C PGM=BPXBATCH,REGION=8M
//STEPLIB DD	DSN=CEE.SCEERUN,DISP=SHR
//STDERR DD	PATH='/u/turbo/bin/mystd.err',
11	PATHOPTS=(OWRONLY,OCREAT,OTRUNC),PATHMODE=SIRWXU
//STDPARM DD	*
SH nohup compr	ess /usr/lib/junk & sleep 1
/*	

Example: The following JCL is similar to the previous example and produces equivalent results but uses PARM= to specify the parameter string:

//jobname	JOB	
//stepname	EXEC	PGM=BPXBATCH,REGION=8M,
11		PARM='SH nohup compress /usr/lib/junk & sleep 1'
//STEPLIB	DD	DSN=CEE.SCEERUN, DISP=SHR
//STDERR	DD	PATH='/u/turbo/bin/mystd.err',
//		PATHOPTS=(OWRONLY,OCREAT,OTRUNC),PATHMODE=SIRWXU

Example: Running a z/OS UNIX executable file or REXX exec in batch

You can also use BPXBATCH to run a z/OS UNIX executable file or REXX exec through MVS batch and redirect the output and error messages to z/OS UNIX files or MVS data sets.

Example: User JAYMC runs an executable file in batch, as follows:

- The program name to be run is /u/jaymc/bin/xparse1.
- STDOUT is to be written to the file /u/jaymc/bin/mystd.out.
- STDERR is to be written to the file /u/jaymc/bin/mystd.err.
- STDIN defaults to /dev/null.

```
//jobname JOB ...
//stepname EXEC PGM=BPXBATCH,REGION=100M,
// PARM='PGM /u/jaymc/bin/xparse1'
//STEPLIB DD DSN=ISFSHR.JAYMC.ISFLOAD,DISP=SHR
//STDOUT DD PATH='/u/jaymc/bin/mystd.out',PATHOPTS=(OWRONLY,OCREAT),
// PATHMODE=SIRWXU
//STDERR DD PATH='/u/jaymc/bin/mystd.err',PATHOPTS=(OWRONLY,OCREAT),
// PATHMODE=SIRWXU
```

Example: This example is very similar to the previous one, except that STDOUT and STDERR are directed to members of an existing PDSE. If you wish to use two members of the same partitioned data set for STDOUT and STDERR output, then you must use a PDSE (not a PDS).

- The program name to be run is /u/jaymc/bin/xparse1.
- STDOUT is to be written to the PDSE member JAYMC.MYSTDLIB(XP10UT).
- STDERR is to be written to the PDSE member JAYMC.MYSTDLIB(XP1ERR).
- STDIN defaults to /dev/null.

//jobname	JOB	
//stepname	EXEC	PGM=BPXBATCH,REGION=100M,
//		PARM='PGM /u/jaymc/bin/xparse1'
//STEPLIB	DD	DSN=ISFSHR.JAYMC.ISFLOAD,DISP=SHR
//STDOUT	DD	DSN=JAYMC.MYSTDLIB(XP10UT),DISP=MOD
//STDERR	DD	DSN=JAYMC.MYSTDLIB(XP1ERR),DISP=MOD

Invoking BPXBATCH from the TSO/E environment

You can use BPXBATCH to run a z/OS UNIX shell command, shell script, or executable file from the TSO/E environment. (For shell commands, however, it may be even easier to use the OSHELL exec, which invokes BPXBATCH. See <u>"OSHELL: Running a shell command from the TSO/E READY prompt" on page</u> 155.)

You can invoke BPXBATCH under TSO/E like this:

BPXBATCH SH|PGM program_name [arg1...argN]

where:

- When SH is specified, *program_name* is the name of a shell command or a file containing a shell script. SH starts a login shell which processes your .profile before running a shell command or shell script. SH is the default; therefore, you can allocate a file containing a shell script to the STDIN ddname, invoke BPXBATCH without any parameters, and the shell script will be invoked.
- When PGM is specified, *program_name* is the name of an executable file that is stored in a z/OS UNIX file. Inadvertent use of a shell script with PGM may result in a process that will not end as expected, and will require use of the kill -9 pid command to force termination.
- You can supply optional arguments, arg1...argN, to program_name.
- You can invoke BPXBATCH without any parameters on the command line and, instead, place the parameter data in a file or data set defined by STDPARM.

Prior to invoking BPXBATCH, you can allocate any of the resources discussed previously, using the TSO/E ALLOCATE command with the following ddnames:

DDname

Description

STDIN

Standard input (see "Defining standard input, output, and error streams for BPXBATCH" on page 146)

STDOUT

Standard output (see <u>"Defining standard input, output, and error streams for BPXBATCH" on page</u> 146)

STDERR

Standard error (see "Defining standard input, output, and error streams for BPXBATCH" on page 146)

STDENV

Environment variable definitions (see "Passing environment variables to BPXBATCH" on page 147)

STDPARM

BPXBATCH parameter data (see "Passing parameter data to BPXBATCH" on page 149)

OSHELL: Running a shell command from the TSO/E READY prompt

The OSHELL REXX exec, shipped in SYS1.SBPXEXEC, invokes BPXBATCH to run non-interactive shell commands from the TSO/E READY prompt. The output is displayed in your TSO/E session.

OSHELL usage notes: Note that:

- 1. With OSHELL, you cannot use a shell command with an & (ampersand) to run it in the background.
- 2. OSHELL cannot be used to invoke an interactive shell command.
- 3. OSHELL creates a temporary file in the **/tmp** directory. The name of the temporary file includes the time, to avoid naming conflicts (for example, **/tmp/userid1.12:33:32.461279.IBM**). The file is deleted when OSHELL completes.

OSHELL examples: For example:

To delete the file **dbtest.c**, user TURBO would enter at the TSO/E READY prompt:

oshell rm -r /u/turbo/testdir/dbtest.c

To display the amount of free space in your file system, you could enter:

oshell df -P

To display information on all accessible processes, you could enter:

oshell ps -ej

Figure 10 on page 156 shows how OSHELL is coded.

```
/* REXX */
parse arg shellcmd
username =,
TRANSLATE(userid(),'abcdefghijklmnopqrstuvwxyz','ABCDEFGHIJKLMNOPQRSTUVWXYZ')
/* Free STDERR just in case it was left allocated
                                                                   */
/*
                                                                   */
msgs = msg('OFF')
"FREE DDNAME(STDERR)"
/ ALLOCATE FILE(STDOUT) PATH('/tmp/"username"."time('L')".IBM') ",
"PATHOPTS(OWRONLY,OCREAT,OEXCL,OTRUNC) PATHMODE(SIRWXU)",
"PATHDISP(DELETE,DELETE)"
IF RC \neg = 0 Then
 DO
    "FREE DDNAME(STDOUT)"
    "ALLOCATE FILE(STDOUT) PATH('/tmp/"username"."time('L')".IBM') ",
    "PATHOPTS(OWRONLY,OCREAT,OEXCL,OTRUNC) PATHMODE(SIRWXU)",
"PATHDISP(DELETE,DELETE)"
   IF RC \neg = 0 Then
     DO
       msgs = msg(msgs)
        /* Allocate must have failed */
       Say ' This REXX exec failed to allocate STDOUT.'
Say ' This REXX exec did not run shell command ' shellcmd
       Say
        RETURN
      END
 END
msgs = msg(msgs)
"BPXBATCH SH "shellcmd
IF RC \neg = 0 Then
   DO
    Say ' RC = ' RC
    Say '
   END
IF RC > 255 Then
   DO
    Say ' Exit Status = ' RC/256
Say ' '
   END
IF (RC ¬= 254) & (RC ¬= 255) THEN
   DO
     "ALLOCATE FILE(out1) DA(*) LRECL(255) RECFM(F) REUSE"
     "OCOPY indd(STDOUT) outdd(out1) TEXT PATHOPTS(OVERRIDE)"
     "FREE DDNAME(out1)"
   FND
"FREE DDNAME(STDOUT)"
```

Figure 10: The OSHELL REXX exec

Using TSO/E REXX for z/OS UNIX processing

You can use a set of z/OS UNIX extensions to TSO/E REXX—host commands and functions—to access kernel callable services. The z/OS UNIX extensions, called *syscall commands*, have names that correspond to the names of the callable services that they invoke—for example, access, chmod, and chown.

You can run a REXX program with z/OS UNIX extensions from MVS, TSO/E, the shell, or a C program. The exec is not portable to an operating system that does not have z/OS UNIX installed.

For more information about the REXX extensions that call z/OS UNIX services, see <u>z/OS Using REXX and</u> <u>z/OS UNIX System Services</u>.

Using the ISPF shell

With the ISPF shell (ISHELL), a user or systems programmer can use ISPF dialogs instead of shell commands to perform many tasks, especially those related to file systems and files. An ordinary user can use the ISPF shell to work with:

- Directories
- Regular files
- FIFO special files
- Symbolic links, including external links

You can also run shell commands, REXX programs, and C programs from the ISPF shell. The ISPF shell can only direct **stdout** and **stderr** to a file in your file system, not to your terminal. If it has any contents, the file is displayed when the command or program completes.

Invoking the ISPF shell

You can invoke the ISPF shell in one of the following ways:

• Type the TSO/E command: ISHELL [-d] [pathname]

See <u>"Entering a TSO/E command" on page 185</u> for information about entering TSO/E commands in TSO/E, the shell, and ISPF.

• Select the ISPF shell from the ISPF menu, if a menu option is installed.

The optional *pathname* parameter specifies the initial path name that you want to appear on the ISHELL main panel.

Example: The following command invokes the ISPF shell and supplies the path name /tmp/ on the ISHELL main panel:

ishell /tmp/

Guidelines: When invoking the ISPF shell, follow these guidelines:

- 1. ISHELL can be invoked with the option –d, which prevents ISHELL from suppressing ISPF severe dialog errors. This will cause ISHELL to terminate on errors. This option should only be used at the direction of an IBM technical support representative.
- 2. The environment variable BPXWISHTZ can be set to a time zone value to have ISHELL use a local time zone that is different than your TZ setting. BPXWISHTZ must be specified in /etc/profile or in .profile. For example, if the TZ setting does not specify GMT, to allow ISHELL users to return to GMT add the following line to etc/profile or .profile:

export BPXWISHTZ=GMT

For more information about using the SMFPRM*xx* parmlib member to specify timeouts, see <u>z/OS UNIX</u> System Services Planning.

3. Since ISHELL contains code to run the TSO commands OGET and OPUT, these commands should not be included in the PLATCMD area of any IKTSOxx member in effect. Doing so will result in a delay in exiting ISHELL, especially if a copy operation has been performed in the ISHELL session. In general, none of the z/OS UNIX TSO commands should be listed as PLATCMD entries. For more information, see z/OS UNIX System Services Planning.

Working in the ISPF shell

Figure 11 on page 158 is the main panel, which you see when you invoke the ISPF shell. The action bar has the following choices:

- File
- Directory

- Special file
- Tools
- File systems
- Options
- Setup
- Help

When you select one of these choices, a pull-down panel displays a list of actions.



In the center of the panel, you see four lines. Here you can type the path name of a file (a directory is a type of file) that you want to work with. It can be the name of an existing file or a new file that you are creating.

In the lower part of the panel, you see a command line. Here you can type an action code, a one-character code that specifies an action that you want to perform on the path name you are working with. For example, D is the action code for "delete" (To familiarize yourself with the action codes, press <F1> on the main panel. On the help panel that is displayed, position your cursor under the highlighted words action code and press <F1>.)

Work in the ISPF shell is a two-step sequence:

- 1. Select an object-the path name of a new or existing file.
- 2. Select an action for that object.

For more about ISPF

For more information about ISPF, see z/OS ISPF User's Guide Vol I.

Using the online help facility

In the ISPF shell, you can get help information for:

- Panels
- Fields on panels
- Highlighted words on panels
Position your cursor on one of those locations and press <F1>.

For more information on the online help facility when you begin working in the ISPF shell, select the Help choice on the action bar and read the information there.

Chapter 12. Performance: Running executable files

Note: This information is directed toward users of the z/OS shell. Most examples pertain to the z/OS shell and not to the tcsh shell.

A process is a collection of threads that execute within an address space, along with the required system resources. A user's login shell is one example of a process.

- The OMVS command creates two processes per login: a process to control the terminal and a process for the login shell.
- rlogin and telnet logins each create two processes: one to control the socket connection to the user, another for the login shell.
- Communications Server logins require only one process per login. Consequently, there is no method for requesting a shared address space for the Communications Server login shell.

Most utilities invoked from the shell command line run in new processes that the shell creates.

There is a system-wide limit on:

- The number of z/OS UNIX processes across the system
- The number of z/OS UNIX processes per user

For a discussion of these limits, see the section on defining system limits in <u>z/OS UNIX System Services</u> Planning.

The shell, and other z/OS UNIX commands and daemons, can assign multiple processes to the same MVS address space; this is called a *shared address space*. Using a shared address space offers these advantages:

- A new process in the same address space can be started more quickly than a new process in another address space.
- A new process in the same address space requires fewer system resources (storage, for example) than a new process in another address space.

For rlogin, the system administrator must update /usr/sbin/inetd.conf by adding -m to the rlogind entry to enabled shared address space. When -m is added, the socket connection process and the login shell process share the same address space.

For the OMVS command, use the SHAREAS keyword to enable shared address space. When the SHAREAS keyword is used, the login shell process is nested in the user's TSO address space. Any other login shells started with the OMVS OPEN subcommand are also nested in the user's TSO address space. (With NOSHAREAS, other login shells started with the OMVS OPEN subcommand will each consume another address space.)

To enable shared address space for the shell, issue the command

export _BPX_SHAREAS=YES

interactively, or place it in your \$HOME/.profile. All simple commands (commands that are run in the foreground and not in a pipeline) will then run in processes nested in the shell's address space. If the

_BPX_SHAREAS variable is not set, or if it is not set to the value YES, the shell creates all processes in separate address spaces. No matter how the shell is started (with or without shared address space enabled), you must set _BPX_SHAREAS=YES if processes started by the shell itself are to run in processes nested in the shell's address space.

User applications can also use shared address spaces. See <u>spawn (BPX1SPN, BPX4SPN) — Spawn a</u> process and <u>attach_exec (BPX1ATX, BPX4ATX) — Attach a z/OS UNIX program</u> in <u>z/OS UNIX System</u> Services Programming: Assembler Callable Services Reference for details.

Some processes cannot execute correctly in a shared address space. For example, if a process needs to reserve z/OS system resources that are common to all processes in a z/OS address space, it must run by itself. If two processes using the same resource attempt to execute concurrently in the same address space, they will compete for these resources, causing at least one of them to fail. When a potential storage shortage is detected, the new processes are created in their own address spaces, even if _BPX_SHAREAS=YES is present in the invoker's environment. For more details about these restrictions, see the descriptions of the spawn() function and the <u>spawn (BPX1SPN, BPX4SPN) — Spawn a process</u> callable service.

Improving shell script performance

You might be able to improve shell script performance by setting the _BPX_SPAWN_SCRIPT environment variable to a value of YES. However, when _BPX_SPAWN_SCRIPT=YES, the behavior will not conform completely to the XPG4 Commands & Utilities specification.

See "Improving the performance of shell scripts" on page 40 for more information.

Chapter 13. Communicating with other users

You can communicate only with users in the same environment you are working in. For example, if you are working in the TSO/E environment, you cannot use MVS facilities to send a message to a user working in the shell.

Shell users who want to exchange messages with other shell users at the same system can use shell commands. Other users may prefer to use TSO/E facilities, in order to be able to exchange messages with all TSO/E users, not just those using the shell.

Within the shell, you can send and receive messages using these shell commands:

- mailx
- mail
- write
- talk
- wall

Alternatively, you can switch into your TSO/E session and send messages to any TSO/E user by using TSO/E facilities, through the OFFICE option of the Information Center Facility (ICF), if it is installed on your system, or through TSO/E commands. You can also receive messages using TSO/E.

If your system has Transmission Control Protocol/Internet Protocol (TCP/IP) or other network management facilities installed, you can log in to the TCP/IP network and send messages to users at other systems.

If your system has UUCP (UNIX-to-UNIX Copy Program) installed and set up, you can use this facility to send files to, or run commands or custom applications at, other sites in the UUCP network.

Using mailx to send and receive mail

You can use the mailx command to send a message to a system-specified mail file. When the shell user receiving the message is ready to read messages, he or she can use mailx to see what messages have arrived and read them.

Administrators and users can customize the behavior of mailx in a number of ways, by selecting variables and setting them in files named /etc/mailx.rc and \$HOME/mail.rc. Some variables apply for the duration of any session; others can be set or reset within a session.

The systems programmer can set up a list of variables (using the set command) in the /etc/mailx.rc file. You can use these values as a default, or you can set up a \$HOME/mail.rc file that sets these variables for your personal use. These variables are described in the mailx command description in <u>z/OS</u> UNIX System Services Command Reference.

You can reset certain variables during a session, or, when entering mailx, specify that the variables in the /etc/mailx.rc file are not to be used.

Steps for sending mail to another user

Before you begin: You need to know that you can send a message to one or more users at a time. The following example is a message sent to several users. The word in italics is output by mailx itself.

```
mailx macneil
Subject: Reminder
Our work group meets today at 10:30.
Let's get together in the library.
~c smitha emilig fabish
~.
```

On the first line, the message is addressed just to macneil. The ~c line adds people who will receive copies of the message.

The ~. line identifies the end of the message and indicates to mailx that you are ready to send it. After you type that line and press <Enter>, the message is sent.

Perform the following steps to send mail to another user

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 \sim c before their login names.

5. To end the message and transmit it, type \sim . and press <Enter>.

You know you are done when the system displays an EOT message.

Sending mail to a distribution list

You can send the same message to multiple users at the same time by using a distribution list.

If you use mailx to send a message, you can specify the *address* of each z/OS UNIX user you want to receive the message. The simplest address is the TSO/E user ID.

Example: To send a message to pfeif, lowell, eliza, and fabish, issue:

mailx pfeif lowell eliza fabish

To send a message to a list of people, you can specify an *address alias* that contains a list of login names.

Example: To set up an alias for the test team, issue:

alias test pfunt lulu detsch naga

Result: When you send a message to the address alias test, it will go to all the login names you specified on the alias command.

Aliases that are entered interactively remain in effect only for the current session. If you want to make the address alias permanent, put the alias command in your **.mailrc** startup file.

Sending a message to an MVS operator

You can use the logger shell command to send a message to an MVS operator.

Example: To send a message to an MVS operator:

logger -d1 Is the tape I requested here yet?

Result: A message is sent to a console with the route code 1.

Receiving mail from other users

The simplest way to read incoming messages is to enter the command mailx. This starts an interactive session that lets you read your mail and perform other actions, such as display new messages and delete 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 "/usr	< :/	xxxxxxx - mail/SMI	Type ? foi THA/":	: Hel 3 me	Lp. essag	ges	3 new		
>N 1 >N 2 >N 3 ?	2	CLIFLWR HOMEBRW ELVIS	·	Thu Thu Thu	Jul Jul Jul	15 15 15	14:28 15:03 16:17	6/93 5/81 6/95	testing lunch plans softball

The first line is the mailx program banner; xxxxxx is information about the version of mailx. As indicated, you can type ? to see a help panel. The second line displays the name of the mailbox being used, /usr/mail/SMITHA/, followed by the number of messages in the mailbox, and their status. Then, you see a list of three messages:

- Number 1 was sent by CLIFLWR and has the subject "testing". It was sent on July 15 at 2:28 PM, 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 PM, and contains 5 lines and 81 characters.
- Number 3 was sent by ELVIS and has the subject "softball". It was sent on July 15 at 4:17 PM, and contains 6 lines and 95 characters.

The user names are all displayed in uppercase.

The question mark (?) is the mail program prompt; it indicates that you can enter mailx subcommands now. Try the subcommand n (next message) to read the messages in sequence:

```
? n
Message 1:
From CLIFLWR Thu Jul 15 14:28
To: SMITHA
Subject: testing
I'm setting up a meeting to test the toolkit
on Monday the 19th at 10AM.
Let me know if you can make it.
?
```

The question mark (?) prompt appears after the displayed message. You can also enter the n subcommand with a number to specify a particular message; for example, n 3 displays the message about softball. Now you can choose what to do with the message: reply to it, save it, or delete it.

Replying to mail

At the question mark (?) prompt, you can use the R (reply to sender) subcommand to reply to a particular message. This is an uppercase R: it differs from the r subcommand, which sends the reply to everyone who sent and received the message. When you give the R subcommand, follow it with the message number. For example:

```
? R 1
To: cliflwr
Subject: Re: testing
Yes, I can make the meeting. where ?
EOT
```

The EOT indicates that your reply has been sent.

Saving and deleting mail

If you exit mail without specifically deleting or saving your messages, the system saves those messages.

To save a message, use the s subcommand and give the name of the file you want to save the message in.

Example: To save the file named *climail*:

s climail

Result: If this is an existing file, the message is appended to it. If the file does not exist, it is created.

To delete a message, use the d subcommand and give the number of the message you want to delete:

? d 1 ?

The mail program deletes message number 1 and returns another ? prompt.

Ending the mailx program

To exit from mailx, use the q (quit) subcommand:

?q \$

The shell prompt indicates that you have left mail and can enter shell commands again.

For more information on mailx, see the <u>mailx</u> command description in <u>z/OS UNIX System Services</u> Command Reference.

Using write to send a message or a file

The write command lets you send a message directly to someone else who is logged on to the system. To determine who is logged on, use the who command. The who command displays information about who is logged on in this form:

BUBBA ttyp0002 Feb 8 09:49

where BUBBA is a login name, ttyp0002 is the terminal, and Feb 8 09:49 is the login time.

The typical format of the write command is:

write user_name

However, if a user is logged in more than once, you can specify *terminal* (in the ttyp form that who returns) rather than *user_name*.

Sending a message: An example

Here is an example of how to send a message, using max as the sender and bubba as the recipient:

write bubba

When max sends a message to bubba, bubba receives a message like this:

Message from max (ttyp002) [Feb 8 15:04] ...

After the system establishes the connection to bubba, it sends two alert characters (usually a beeping sound) to max's terminal to indicate that it is ready to send a message. max can then type a message, which appears on bubba's terminal. If a message is more than one line, each time you press <Enter> a line is sent to bubba's terminal.

Ending a message

To end a message, enter <EscChar-D> for end-of-file or <EscChar-C> for an interrupt. When write receives an end-of-message indicator, it displays an EOF message on the other user's screen and breaks the connection.

When your message is completed, the other user can reply to your message with

write your_user_name

However, if both of you are trying to write on each other's terminal at the same time, the messages may get interleaved on your screens, making them difficult to read. For two-way conversations, use talk instead of write. For more information about talk, see <u>"Using talk for an online conversation" on page</u> 167.

Sending a file

You can add the output of a command to a message that you are writing. To do this, start a line with an exclamation mark (!) and put a standard shell command on the rest of that line. write calls your shell to execute the command, and sends the standard output (**stdout**) from the command to the other user. The other user does not see the command itself or any input to the command. For example, you might write:

```
Here is what my file contains:
!cat file1
```

The contents of **file1** are displayed on the other user's screen.

Using talk for an online conversation

talk lets you start up a two-way conversation with someone else logged in to the system. However, talk is available only if you access the shell with rlogin or telnet or the Communications Server, because it requires raw mode.

The typical format of the talk command is:

talk user_name

However, if a user is logged in more than once, you can specify *terminal* (in the form ttyp that who returns) rather than *user_name*.

Beginning a conversation: An example

Here is an example of how to begin a conversation with talk, using **max** as the person starting a conversation with **bubba**. Here **max** begins by typing:

talk bubba

bubba receives a message like this:

```
Message from max.
talk: connection requested by max
talk: respond with: talk max
```

To set up the two-way connection, **bubba** must enter:

talk max

After this connection has been established, the two can type simultaneously.

Viewing the conversation

talk displays incoming messages from the other person in one part of the screen and your outgoing messages in another part of the screen.

Some terminals may not be able to split the screen into parts in this way. Depending on the terminal type, talk may try to simulate this effect. However, it may not be possible for both users to enter messages simultaneously.

Using wall to broadcast messages

A superuser can use the wall command to send a message to all logged in shell users:

wall [message]

If the message is omitted from the command line, the user will receive two beeps as a prompt to enter the message. You input the message, pressing enter after each line, and when done inputting the entire message, enter end-of-file or an interrupt (typically, <EscChar-D> for end-of-file or <EscChar-C> for an interrupt).

The user of wall should be a superuser. This ensures that the user is permitted to write to all the users that are logged on. If a user who is not a superuser attempts to use wall to broadcast a message, some writes will fail and those users will not receive the message.

Users who are sent a broadcast message will receive a beep announcing the message, and a message in the form:

```
Broadcast Message from SWER@AQFT (ttyp0006) at 10:43:54 (EDT5EST) ...
This is the text of the message line1
This is line2
```

For more information on the wall command, see the <u>wall</u> command description in <u>z/OS UNIX System</u> Services Command Reference.

Controlling messages and online conversations

You can use the mesg command to control whether other users can send messages to your terminal with talk, write, or similar commands.

To let other people send you messages, issue:

mesg y

To tell the system not to let other people send you messages, issue:

mesg n

To display the current setting without changing it, issue:

mesg

Using the UUCP network

If your system administrator has UUCP (UNIX-to-UNIX Copy Program) set up to communicate with remote sites, you can use this facility to send or retrieve files, or to run commands or custom applications at other sites in the UUCP network. To send or retrieve files from remote sites, use the uucp command; this causes a file transfer request to be queued. Depending on how your system is set up, a file transfer request may be processed immediately or later at a scheduled time.

UUCP provides the uucp command, which schedules files to be exchanged with other UUCP systems, and the uux command, which schedules commands to be executed by other UUCP systems. However, the uucp and uux commands do not cause any files to be exchanged or commands to be executed. For this, UUCP provides two daemons called uucico and uuxqt, which establish communication sessions, transfer data, and execute commands according to the requests scheduled by uucp and uux.

The commands that you use with UUCP are:

uucp

Copy files between remote systems

uuname

Display a list of UUCP systems

uupick

Manage files sent to you via uuto

uustat

Display the status of pending UUCP transfers

uuto

Copy files to users on remote systems

uux

Request command execution on remote systems

Tip: uucp, uuto, and uupick do not convert file data to or from EBCDIC. The sending and/or receiving user must convert file data if two systems have different codesets. You can use the iconv command to do this.

Transferring a file to a remote site

To transfer a file to a remote site, use the uucp command or the uuto command.

Using uucp to transfer files

uucp automatically handles text and binary files. When a file is transferred by uucp to another site, it is put in the public UUCP directory—by default, this is **/usr/spool/uucppublic**.

1. You need to know the name of the remote site. To list the remote sites that have been configured, type:

uuname

The sites are listed, one per line.

2. Copy the file to the other site.

To make file transfers easier, you can use a special character in pathnames for the public UUCP directory. When tilde (~) is written as the first directory in a destination path name, the ~/ stands for the public UUCP directory. You can specify the public UUCP directory with the pathname ~/.

For example, to copy the file memo1.pay in your current directory to the public directory on the site named north, type:

uucp memo1.pay north!~/memo1.pay

File transfers may not get processed immediately. If there is any chance that the file that is to be sent will not be available later, use the -C option on the uucp command to immediately copy the file to the uucp spool directory. This ensures that the file is available later when the file transfer occurs.

Using uuto to transfer files

uuto is a simplified method of invoking uucp, and it also handles text and binary files automatically. When a file is transferred by uuto to another site, it is put in the **receive/usr** subdirectory of the public UUCP directory. Within the **receive** subdirectory, each user on the local system has a subdirectory. For example, a file for user stiert would be transferred to **/usr/spool/uucppublic/receive/stiert**.

1. You need to know the name of the remote site. To list the remote sites that have been configured, type:

uuname

The sites are listed, one per line.

2. Copy the file to the other site. For example, to copy the file memo1. pay in your current directory to the public directory on the site named north, type:

uuto memo1.pay north!nuucp

The recipient is notified by mail when the file arrives. To get the file, the recipient should use the uupick command. See <u>"Working with your files in the public directory" on page 171</u> for information on how to use the uupick command.

Transferring multiple files to a remote site

You can use uucp to transfer more than one file, specifying the files by name or by using wildcards. To send more than one file, you must specify a directory as the destination, not a file name. To do this, end the destination pathname with a slash (/).

For example, to send the files **jan.wks**, **feb.wks**, **mar.wks**, and **memo1.txt**, to the directory **receive** at the north site, type:

uucp *.wks memo1.txt north!~/receive/

The trailing slash (/) shows that **receive** is a directory.

You can send an entire directory, by specifying the contents of the directory with a wildcard.

Transferring a file to the local public directory

You may want to put a file in your local public directory so that others can access it there. To specify the public directory in a local pathname, put single quotation marks around the pathname so that the shell does not treat the tilde as your home directory. (For more information on how the shell interprets a tilde in file names, see <u>"Characters used in file names" on page 71.</u>)

Example: To copy that file to your own UUCP public directory, issue:

```
uucp memo1.pay '~/memo1.pay'
```

Notification of transfer

If you want to be certain that a file has been transferred, or if you want someone at the remote site to know that the file has arrived, you can use the -m and -n options on the uucp command, or the -m option on the uuto command.

• With uucp -m or uuto -m, as soon as the file is successfully transferred, you receive a mail message. You can use mailx to read the message. The first line describes the file transfer request, and the second line describes the result. For example, it might look like this:

```
REQUEST: home!/usr/spool/uucppublic/memo1.txt north!/usr/spool/uucppublic/memo1.txt (SYSTEM north) copy successful
```

• With uucp -n *name*, if you are transferring a file to a remote site, you can specify the login name of the person at the remote site to be notified when the file is transferred. That person can read the notification message using mailx.

Permissions

Each site in a UUCP network has a Permissions file that is used to control the access that remote systems have to data and programs on the local system. This file is used to specify, among other options, the areas in the file system that a remote system can read or write from, the commands that the remote system can run on the local system, and a different public directory than the default. Those options are specified as:

READ

Indicates which directories can be read. By default, this is the home directory of user uucp (/usr/spool/uucppublic).

WRITE

Indicates which directories uucico can write to. By default, this is /usr/spool/uucppublic, the home directory of user *uucp*.

NOREAD

Indicates that files in the specified directories cannot be read. If a directory is specified by both READ and NOREAD, files in that directory cannot be read. The public directory can always be read (even if specified on NOREAD).

NOWRITE

Indicates that files in the specified directories cannot be written to. If a directory is specified by both WRITE and NOWRITE, files in that directory cannot be written to. The public directory can always be written to (even if specified on NOWRITE).

PUBDIR

Indicates the public directory. By default, this is the home directory of user *uucp* (/usr/spool/uucppublic).

COMMANDS

Indicates the commands that the remote system can execute on your system. If more than one command is specified, the command names are separated with a colon (:). For example, COMMANDS=uucp:ls. If all commands are prohibited, the COMMANDS option is not used.

For a full description of all the Permissions file options, see the <u>permissions file</u> in <u>z/OS UNIX System</u> Services Planning.

Transferring a file from a remote site

To copy a file from a remote site, your site must have read permissions on the file. Normally your site would have read permissions only on the public UUCP directory and its subdirectories.

For example, say you want to copy the program **pages** from **programs**, a subdirectory of the remote site's public UUCP directory, to your public UUCP directory.

To retrieve the file, you would enter this command:

uucp south!~/programs/pages '~/pages'

where south is the remote site.

For more information about the uucp command, see z/OS UNIX System Services Command Reference.

Checking a file's transfer status

To check the status of pending transfer requests, use the uustat command. You can specify options to display the status of transfers for a particular job ID or user ID.

To display completed file transfer attempts, use the uulog command. To see the record of completed file transfer attempts and connections by site, type:

uulog -s site

where site is the name of the remote site.

Working with your files in the public directory

All users have read access to the UUCP public directory. When you have a file in the public directory, you can use the cp command to copy the file or the mv command to move the file. If the sender uses the -n option on uucp, you are notified when the file is placed in the public directory.

Files sent to you with the uuto command are automatically placed in the **receive** subdirectory. You can use uupick to manage files in the **receive** subdirectory of the UUCP public directory. If **receive** is specified as the target directory on the uucp command, you can use uupick to manage the files.

Within the **receive** subdirectory, each user on the local system has a subdirectory.

To check your public UUCP directory for files sent to you by the uuto command, type uupick. For each file or directory found, uupick prompts you with a message and then you specify how that entry should be handled. For example, for a file, it might display:

from south: file memo2.txt ?

In response, you could type d to delete the file, or m to move the file into your current working directory, or m /mydir/tmp to put it in the directory **/mydir/tmp**.

For more information about the uupick command, see *z/OS UNIX System Services Command Reference*.

Running a command on a remote site

You can use the uux command to run commands on remote sites, but they cannot be interactive commands such as vi. You must have a working UUCP connection and permission to execute commands on the remote site.

Using a remote file as an argument

To ask south to print the file **south!/schedule/january** using the 1p command, you would type:

uux 'south!lp' '/schedule/january'

where **/schedule/january** is the name of the file on south to be printed. In general, if no site is specified on the arguments for the remote command, uux assumes the command is on the site running the command. You must specify full pathnames for files in uux commands. As a general rule, enclose all arguments to uux in single quotation marks to prevent the shell from interpreting them.

Using a local file as an argument

To ask south to print the local file **/schedule/january** using the 1p command, you would type:

uux south!lp !/schedule/january

uux sends a copy of the file for printing; after the remote command has run, the copy is removed.

Using TSO/E to send or receive mail

You can use the TSO/E panel facilities or TRANSMIT and RECEIVE commands to communicate with any TSO/E user (including z/OS UNIX users). If you use TSO/E to send a message, your correspondent must use TSO/E to receive it.

Sending a message

You can use the TSO/E Information Center Facility (ICF), if installed, or TSO/E commands to send a message. For example, to send a short message (with no more than 115 characters), you can switch to TSO/E command mode and enter:

SEND 'Have to go home to take my cat to the vet' USER(alice)

You use SEND for messages to people on the same system as you.

For a longer message, or a message to someone on a different system, you could use:

TRANSMIT dallas.alice

where dallas.alice identifies the person to receive the message: dallas is the ID of the MVS system (known as a *node* in the network) where the person works, and alice is the person's user ID. The system then prompts you to enter the message.

Sending a message to a distribution list

You can use the TSO/E ICF, if installed, or the TSO/E TRANSMIT command to send a message to a distribution list. You set up a distribution list by specifying a nickname entry in the NAMES data set that contains a list of names or nicknames you want the message sent to.

Example: If you have set up the nickname test for a distribution list, issue:

transmit test

Result: The system displays a screen for input. Type your message and press <F3> to send it.

Sending a message to an MVS operator

To send a message to a specific MVS operator, you must know the operator's route code and specify it in the OPERATOR operand.

Example: Issue:

SEND 'Are the tapes I wanted from the library here yet?' OPERATOR(7)

You can also send a message to a specific operator console by using the CN operand. A console name or ID is defined at your enterprise.

Example: To send a message to the operator console named TAPELIB, issue:

send 'please send the tapes to the floor.' CN(TAPELIB)

Receiving mail from other users

How and whether you are notified when TSO/E messages are received by the system depends on how your TSO/E system is set up:

- You may be notified when you log on or as messages arrive.
- You may have to enter a RECEIVE command periodically to see if a message has arrived.

Unless the messages are automatically displayed when you log on, you enter a RECEIVE command to see your currently unread messages. For more information on TSO/E mail and messaging, see <u>z/OS TSO/E</u> User's Guide.

Receiving messages from other systems

TSO/E users can receive messages from other systems through the TSO/E message interface. Receiving a message from a user on another system is the same as receiving one from a user on the same system.

Part 2. The z/OS UNIX file system

These topics discuss tasks involved with the z/OS UNIX file system.

Chapter 14. An introduction to the z/OS UNIX file system

z/OS UNIX files are organized in a hierarchy, as in a UNIX system. All files are members of a *directory*, and each directory is in turn a member of another directory at a higher level in the hierarchy. The highest level of the hierarchy is the *root directory*.

The root file system and mountable file systems

Taken as a whole, the *file system* is the entire set of directories and files, consisting of all files shipped with the product and all those created by the systems programmer and users. The systems programmer (superuser) defines the *root file system*; subsequently, a user with mount authority can mount other mountable file systems on directories within the file hierarchy. (See the section on <u>mounting file systems</u> in <u>z/OS UNIX System Services Planning</u>.) Altogether, the root file system and mountable file systems comprise the file hierarchy used by shell users and applications.

After installation of z/OS, the end user's logical view of the file system is as shown in Figure 12 on page 177.



Figure 12: End user's logical view of the file system

System programmers need to know that the illustration of directories in Figure 12 on page 177 is not a true representation of file systems. The file system, as installed through ServerPac or CBPDO, consists of /dev, /tmp, /var, and /etc symbolic links that point to the /dev, /tmp, /var, and /etc directories, as demonstrated in Figure 13 on page 177.



Figure 13: Organization of the file system

Here SYSTEM is a data set that contains directories which are used as mount points, specifically for the / etc, /var, /tmp, and /dev file systems. IBM requires that you mount /etc, /var, /dev, and /tmp in separate data sets.

The z/OS shells and utilities typically impose a *line orientation* on the byte-oriented files. A *line* is a stream of bytes terminated with a <newline> character. A line terminated by a <newline> character is sometimes referred to as a record. So, there is a single <newline> character between every pair of adjacent records. Text files use the <newline> character to delimit lines; binary files do not.

Several types of file systems can be mounted within the file hierarchy:

- z/OS File System (zFS): zFS is the strategic file system for z/OS UNIX and continues to be enhanced to provide superior performance, reliability, and data integrity. zFS file systems can be mounted into the z/OS UNIX hierarchy along with other local (or remote) file system types (for example, HFS, TFS, AUTOMNT and NFS). Also, zFS can support access control lists (ACLs). For more information, see <u>z/OS</u> *Distributed File Service zFS Administration* and *z/OS Migration*.
- Hierarchical File System (HFS): In HFS, the entire file hierarchy is a collection of hierarchical file system data sets (HFS data sets). Each HFS data set is a mountable file system. DFSMS facilities are used to manage an HFS data set, and DFSMS Hierarchical Storage Manager (DFSMShsm) is used to back up and restore an HFS data set.
- Network File System (NFS): Using NFS client on z/OS UNIX System Services, you can mount a file system, directory, or file from any system with an NFS server within your user directory. You can edit or browse the files. For more information, see <u>"Using the Network File System feature" on page 186</u> and z/OS Network File System Guide and Reference.
- Distributed File System (DFS): DFS joins the local file systems of several file server machines, making the files equally available to all DFS client machines. DFS allows users to access and share files stored on a file server anywhere in the network, without having to consider the physical location of the file. For more information, see *z*/OS DFSMS Using Data Sets.
- Temporary File System (TFS): The TFS is an in-memory physical file system that delivers high-speed I/O. To take advantage of that, the systems programmer (superuser) can mount a TFS over the /tmp directory so it can be used as a high-speed file system for temporary files. (Normally, the TFS is the file system that is mounted instead of the HFS if the kernel is started in minimum setup mode.)

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 contained in that directory (although their actual locations in physical storage are managed by the operating system).

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 12 on page 177, the slash (/) symbol at the top represents the root directory, which all other directories are descended from. There are ten directories branching from the root. Each of these directories, in turn, has its own system of subdirectories and files. For example, localedef is a subdirectory in the directory /usr/lib/nls.

When you first enter the z/OS shell, you are automatically placed in your *home directory*, which is defined when your user ID is defined.

Files

In addition to directories, there are four other types of files that can exist in the file system:

• A *regular file* is an identifiable (named) unit of text or binary data information. A file can be C source code, a list of names or places, a printer-formatted document, a string of numbers organized in a certain way, an employee record containing smaller information units in fields, a memo, and many other

possible things. A user or an application program must understand how to access and use the individual increments of information (such as employee record fields) within a file.

- A character special file defines one of the following:
 - A terminal (/dev/ptypnnnn and /dev/ttypnnnn).
 - The default controlling terminal for a process (/dev/tty).
 - A null file (/dev/null). Data written to this file is discarded; hence, it is known as the bit bucket. This file is always empty for reading.
 - A zero file (/dev/zero). Data written to this file is discarded and binary zeros are supplied for any amount read from it.
 - The random number files (/dev/random and /dev/urandom). These files provide random numbers for cryptographic purposes.
 - A file descriptor file (/dev/fdn or /dev/fd/n).
 - A system console file (/dev/console). Data that is written to this file is sent to the console using a write-to-operator (WTO) that displays the data on the system console.
 - A UNIX domain socket name file. This is a path name that specifies the socket address for a UNIX domain socket. The path name is assigned by the application programmer; there is no convention for the name. The operating system creates the file.
 - A Communications Server remote tty file (for example, rtynnnn) that corresponds to the requesting terminal on the originating Communications Server node. The name is assigned by the Communications Server administrator.
 - The Communications Server character special file (/dev/ocsadmin) that supports ioctl functions for Communications Server administrative functions.

Character special files are dynamically created by the operating system when they are first referenced. However, they can also be explicitly created by a superuser (for instance, in order to assign different permissions).

- A FIFO special file is a file that is 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 contains the path name for another file, in essence a reference to the original file. Only the original path name is the real name of the original file. You can create a symbolic link to a file or a directory. These are symbolic links: /etc, /tmp, /dev, and /var.

An *external link* is a type of symbolic link, a link to an object outside of the z/OS UNIX file system. Typically, it contains the name of an MVS data set.

Users and programs create regular files, FIFO special files, symbolic links, and external links.

Files not in the file system

There are two types of unnamed files that you might be aware of, but that do not exist in the file system:

Unnamed pipes: A program creates a pipe with the pipe() function. A pipe typically sends data from one process to another; the two ends of a pipe can be used in a single program task. A pipe does not have a name in the file system, and it vanishes when the last process that is using it closes it.

Sockets: A program creates a socket with the socket() function. A socket is a method of communication between two processes that allows communication in two directions, in contrast to a pipe, which allows communication in only one direction. The processes using a socket can be on the same system or on different systems in the same network.

Comparison between MVS data sets and the z/OS UNIX file system

In Figure 14 on page 180, you see that:

• The MVS master catalog is analogous to the root directory in a z/OS UNIX file system.

• The user prefix assigned to MVS data sets is an organizer analogous to a user directory (/u/smitha) in the file system. Typically, one user owns all the data sets whose names begin with his user prefix. For example, the data sets belonging to the TSO/E user ID SMITHA all begin with the prefix SMITHA. There could be data sets named SMITHA.TEST1.C, SMITHA.TEST2.C, SMITHA.TEST1.LIST, and SMITHA.TEST2.LIST.

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

• Of the various types of MVS data sets, a partitioned data set (PDS) is most akin to a user directory in the file system. In a partitioned data set such as SMITHA.TEST1.C, you could have members PGMA, PGMB, and so on—for example, SMITHA.TEST1.C(PGMA) and SMITHA.TEST1.C(PGMB). Likewise, a subdirectory such as /u/smitha/test1 can hold many files, such as pgma.c, pgmb.c, and so on.



Figure 14: Comparison of MVS data sets and the z/OS UNIX file system

Sharing files between LPARs

z/OS UNIX files cannot reside on a DASD that is shared in read/write mode between LPARs. However, you can share z/OS UNIX files (both zFS files and HFS files) between LPARs when you use the shared file system capability provided by z/OS UNIX. To share only zFS files, you can use the sysplex support found in zFS.

Executable modules in the file system

You can have an executable module in the z/OS UNIX file system. To run a shell script or executable, a user must have read and execute permissions to the file. Use chmod to set the permissions.

- For frequently used programs in the file system, you can use the chmod command to set the *sticky bit*. This reduces I/O and improves performance. When the bit is set on, z/OS UNIX searches for the program in the user's STEPLIB, the link pack area, or the link list concatenation. For further information, see "Using a symbolic mode to specify permissions" on page 219.
- The extattr command is used to set, reset and display extended attributes for files to allow executable files to be marked so they run APF authorized, as a program controlled executable, or not in a shared address space.

The 1s shell command has an option that displays these attributes:

-E

Displays extended attributes for regular files:

Program runs APF authorized if linked AC=1

р

а

Program is considered program controlled

S

Program runs in a shared address space

Attribute not set

When the extattr attribute 1 is set (+1) on an executable program file, it will be loaded from the shared library region.

- You can copy executable modules between z/OS UNIX and the file system. For more information about how to do this, see <u>"Copying executable modules between MVS data sets and the z/OS UNIX file</u> system" on page 264.
- For information about how to set up a STEPLIB environment for an executable file, see <u>"Building a</u> STEPLIB environment: The STEPLIB environment variable" on page 45.

For more information about the <u>ls</u> and <u>extattr</u> shell commands, see <u>z/OS UNIX System Services Command</u> Reference.

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.

Instead of using the absolute path name with shell 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 is

recommended. Using DBCS data in these names is not recommended because unpredictable results might occur.

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 an absolute path name

In some situations, an absolute path name is required. <u>Table 10 on page 182</u> shows that job control language (JCL) and some TSO/E commands require an absolute path name and that they require an MVS data set name to be specified in a certain way. In these situations, the maximum length of the absolute path name is 255 characters.

Commands	Path name	Dataset name
JCL	Absolute, in single quotation marks	Fully qualified (no quotation marks needed).
ALLOCATE command	Absolute, in single quotation marks	Fully qualified in single quotation marks. If specified without quotation marks, the TSO/E prefix is added to the data set name. Normally the TSO/E prefix is the TSO/E user ID (this can be hanged with the PROFILE PREFIX() command).
OEDIT and OBROWSE commands	Absolute, unless you are working in your current directory	Not applicable
OPUT, OGET commands	Absolute (unless you are working in your home directory), in single quotation marks	Fully qualified in single quotation marks. If specified without quotation marks, the TSO/E prefix is added to the data set name. Normally the TSO/E prefix is the TSO/E user ID. The prefix can be changed with the PROFILE PREFIX() command.
OPUTX, OGETX commands	Absolute (unless you are working in your home directory)	Fully qualified in single quotation marks. If specified without quotation marks, the TSO/E prefix is added to the data set name. Normally the TSO/E prefix is the TSO/E user ID. The prefix can be changed with the PROFILE PREFIX() command.

Table 10: Requirements for absolute path names

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.

Note: An external link is a type of symbolic link that refers to an object outside of the hierarchical file system. As used by the Network File System feature, an external link refers to an MVS data set name.

Symbolic and external links with a sticky bit

The following behavior applies to DLLs and all forms of spawn() and exec(). What applies for exec() also applies for all forms of module loading.

External links

exec() does a stat() on the passed file name. stat() does the search, not exec(). If the file name is an external link, then stat() fails with a unique reason code which causes exec() to read the external link. If the external link name is a valid PDS member name (that is, 1 - 8 alphanumeric or special characters), then exec() attempts to locate the module in the MVS search order. If it cannot find the module, exec() fails.

The external link is normally used when you want to set the sticky bit on for a file name which is longer than eight characters or contains characters that are unacceptable for a PDS member name.

• Symbolic links

If the file name you specify is a symbolic link and exec() sees the sticky bit on, then it truncates any dot qualifiers. As long as the base file name is an acceptable PDS member name, the need to set up links in order to get exec() to go to the MVS search order should not be an issue.

For example, if you have a file named java.jll, when you set the sticky bit on, exec() attempts to load a member named JAVA. If exec() cannot find JAVA, it reverts to using the java.jll file in the file system.

The important thing to understand is that exec() never sees the name that the symbolic link resolves to, even though it can see the stat() data for the final file.

Example: If you define /u/user1/name1 as a symbolic link to /u/user1/name2 and then invoke *name1*, the following occurs:

- 1. The shell will spawn name1.
- 2. spawn() will access the file for *name1*, unaware that there is a symbolic link already established. It will access the *name2* file by its underlying vnode, not by the *name2* handle.
- 3. If the sticky bit is on for the *name2* file, spawn() will do the MVS search for *name1* (the only name it has to work with).

Command differences with symbolic links

Certain directories like /etc, /dev, /tmp, and /var are converted to symbolic links. Some shell commands have minor technical differences when they refer to symbolic links instead of regular files or directories. For example, 1s does not follow symbolic links by default.

In order to follow symbolic links, you must specify 1s -L or provide a trailing slash. For example, 1s -L /etc and 1s /etc/ both display the files in the directory that the /etc symbolic link points to.

Other shell commands that have differences due to symbolic links are du, find, pax,rm, and tar.

While these behavioral changes should be minor, users can tailor command defaults by creating aliases for the shell command. For example, if you want 1s to follow symbolic links, you could issue the command alias 1s="1s -L". Aliases are typically defined in the users' ENV file.

Note: After this alias has been established, 1s will follow all symbolic links.

An administrator can put alias commands in /etc/profile that could affect all users' login shells. IBM does not recommend this, because changing the default behavior in /etc/profile might produce unexpected results in shell scripts or by shell users.

Using commands to work with directories and files

There are numerous shell commands you can use to create and work with directories and files. See the z/OS shell summary section in z/OS UNIX System Services Command Reference for a list of them.

To get online help for using the shell commands, you can use the man command.

You can also use TSO/E commands to do certain tasks with the file system. Some of these are tasks that UNIX users traditionally perform while in the shell.

Command

Task

ISHELL

Invoke the ISPF shell. This is a panel interface for performing many user and administrator tasks. For more information, see "Using the ISPF shell" on page 157.

MKDIR

Create a directory. Unlike the mkdir shell command, this command does not create intermediate directories in a path name if they do not exist.

MKNOD

Make a character special file. To use this command, you must be a superuser.

MOUNT

Add a mountable file system to the file hierarchy. To use this command, you must have mount authority. (See the section on mounting file systems in *z/OS UNIX System Services Planning*.)

OBROWSE

Browse (read but not update) a z/OS UNIX file using the ISPF full-screen browse facility.

OCOPY

Copy data between sequential data sets, or PDS and PDSE members, and z/OS UNIX files.

OEDIT

Create or edit text using the ISPF editor.

OGET

Copy a z/OS UNIX file to an MVS sequential data set or partitioned data set member. You can specify text or binary data, and select code page conversion.

OGETX

Copy one or many files from a directory to a partitioned data set, a PDS/E, or a sequential data set. You can specify text or binary data, select code page conversion, allow a copy from lowercase file names, and delete one or all suffixes from the file names when they become PDS member names.

OPUT

Copy an MVS sequential data set or partitioned data set member to a z/OS UNIX file. You can specify text or binary data, and select code page conversion.

OPUTX

Copy one or many members from a partitioned data set, PDS/E, or a sequential data set to a directory. You can specify text or binary data, select code page conversion, specify a copy to lowercase file names, and append a suffix to the member names when they become file names.

OSTEPLIB

Build a list of files that are sanctioned as valid step libraries for programs that have the set-user-ID or set-group-ID bit set. To use this command, you must be a superuser.

UNMOUNT (or UMOUNT)

Remove a file system from the file hierarchy. To use this command, you must have mount authority. (See the section on mounting file systems in *z/OS UNIX System Services Planning*.)

For information about existing TSO/E commands that you might commonly use, see <u>z/OS TSO/E Command</u> Reference.

To get online help for TSO/E commands, you can use either the TSO/E HELP command. See <u>"Entering a TSO/E command" on page 185</u> for information about entering TSO/E commands in TSO/E, the shell, and ISPF.

Entering a TSO/E command

How you can enter a TSO/E command depends on whether you are using the OMVS terminal interface or the asynchronous terminal interface you get with rlogin, telnet, or the Communications Server.

On the OMVS terminal interface, you can enter a TSO/E command:

- At the TSO/E READY prompt.
- In the shell, using the tso shell command. For more information about this command, see <u>z/OS UNIX</u> System Services Command Reference.
- In the shell, by typing a TSO/E command at the shell prompt and pressing the TSO function key to run it.
- On an ISPF panel.

CA CA

CAUTION: You need to be aware of two things about entering TSO/E commands in ISPF:

1. On most ISPF panels, you must type TSO before the name of the TSO/E command; for example,

TSO OBROWSE 3 fopen

However, on the **TSO Command Processor** panel (ISPF option 6), you can just enter the name of the TSO/E command, unless the command exists in both ISPF and TSO (for example, HELP or PRINT).

2. On most ISPF panels, ISPF folds what you type to uppercase. ISPF folds lowercase or mixedcase file names to uppercase, even if they are enclosed in single quotation marks. However, the **TSO Command Processor** panel (ISPF option 6) processes what you enter exactly as it is typed —mixed case, uppercase, or lowercase.

On an asynchronous terminal interface, you can enter TSO/E commands in the shell, using the \pm so shell command. For more information about this command, see the \pm so command in <u>z/OS UNIX System</u> Services Command Reference.

Using a relative path name on TSO/E commands

If you run a TSO/E command by using the OMVS TSO subcommand or function key or the tso -o command, the TSO/E command runs in your TSO/E address space. The working directory of your TSO/E address space is typically your home directory. Therefore, if you specify a relative path name on a TSO/E command, the system searches for it in your home directory, even if you are working in a different directory.

If you run a TSO/E command by using the tso -t command, it runs in its own process. If you run the command using a relative path name, the system searches for it in your working directory.

Finding the data set that contains a file

To determine which data set (file system) contains a file, use the **df** shell command. You can use **df** to find the data set (file system) that contains your current working directory, or **df** *file name* to find the data set of another file.

Using the ISPF shell to work with directories and files

If you are a user with an MVS background, you may prefer to use the ISPF shell panel interface instead of shell commands or TSO/E commands to work with the file system. The ISPF shell also provides the administrator with a panel interface for setting up users for z/OS UNIX access, for setting up the root file system, and for mounting and unmounting a file system. For more information about the ISPF shell, see "Using the ISPF shell" on page 157.

Using the Network File System feature

Using the Network File System feature, you can mount z/OS UNIX files on an empty directory at your workstation.

To access the z/OS UNIX files, you first enter the mvslogin command, which gives you permission to use NFS.

Then you enter the mount command to make a connection between a mount point on your local file system and a directory or file in the z/OS UNIX file system. After a directory is mounted, you can create, delete, read, or write to a file in or below that directory in the file hierarchy; generally, you can treat a file in or below that directory as a member of your own workstation file system.

- For text files, the Network File System feature handles conversion between the EBCDIC code page used in the z/OS shell and the ASCII code page used at your workstation.
- RACF checks the authority of a workstation user to access z/OS UNIX files on the host. This is based on the authority of the MVS user ID specified on the mvslogin command.

For more information, consult the appropriate Network File System documentation.

External links

An external link is a type of symbolic link that you can use to associate an MVS data set or PDS member with a z/OS UNIX path name. The external link lets the NFS client user transparently access an MVS data set using a path name. A program using the exec() family of functions or the exec, loadhfs, or spawn callable services can also access an MVS data set using an external link.

The data set appears in a mounted z/OS UNIX directory with z/OS UNIX files. If you are working with both MVS data sets and z/OS UNIX files on the workstation, with an external link you can have one directory for both the data sets and the files—for example, /host, instead of /host/ds for the data sets and / host/hfs for the files.

For information about how to create an external link when working at the host, see <u>"Creating an external</u> link" on page 204.

Security for the file system

The security facility is assumed to be the Resource Access Control Facility (RACF). You could use an equivalent security product.

The file system and power failures

If the power fails, you might lose recent data that is still buffered, but the file system structures, directories, inodes and such, will not be damaged. A shadow writing technique is used to ensure that structural changes are always committed automatically. The z/OS UNIX file system does its own repair, as needed, on each mount of a file system. This is based on records it keeps of changes in progress.

There is no fsck command and the z/OS UNIX file system was designed so that this is not needed. The fsck utility generally ensures structural integrity, not data integrity.

Of course, there is always a possibility that user data, critical file system data, or the media can be damaged, so prudent backup procedures are always warranted.

Chapter 15. Converting files between code pages

Enhanced ASCII and Unicode Services make porting applications to z/OS UNIX easier by providing conversion from ASCII to EBCDIC.

Enhanced ASCII

Enhanced ASCII enables users to deal with files that are in both ASCII and EBCDIC format. z/OS is an EBCDIC platform. The z/OS UNIX shells and utilities are configured as EBCDIC programs. That is, characters are coded in the EBCDIC code set. Before z/OS Version 1 Release 2, applications that ran on z/OS UNIX had to exist in EBCDIC form, and expected text data to be stored in EBCDIC form. If you wanted to convert files from EBCDIC to ASCII or ASCII to EBCDIC, you needed to use iconv. With Enhanced ASCII, you can deal with applications and their data in your choice of ASCII or EBCDIC code sets. z/OS UNIX still operates as an EBCDIC system, but it can automatically convert the data from ASCII to EBCDIC and back as necessary to complete commands and tasks.

File tagging in Enhanced ASCII

Enhanced ASCII provides support for file tagging. File tags are used to identify the code set of text data within files. When Enhanced ASCII functionality is enabled, z/OS UNIX needs to know whether files are encoded as ASCII or EBCDIC. The file tag provides this data. If no file tag exists on a particular file, that file is treated as an EBCDIC file. Setting a file tag does not force automatic code set conversion but allows it to take place when automatic code set conversion is enabled. For additional information on automatic code set conversion, see "Automatic code set conversion" on page 190.

Note: An entire file system can be mounted such that untagged files and new files created within the file system are treated as tagged while the mount option is in effect.

z/OS utilities provide options to manage tags on UNIX files. The chtag command allows you to set, modify, remove, or display information in a file tag. In this example:

chtag -t -c IS08859-1 christmas.songs

the file christmas.songs is tagged as an ASCII file. ISO8859–1 is the code set for ASCII. In this example:

chtag -t -c IBM-1047 christmas.recipes

the file christmas.recipes is tagged as an EBCDIC file. IBM-1047 is the code set for EBCDIC.

The ls command with option -T and the command chtag -p display information about the file text and codeset tags. For more information about the chtag and ls commands, see <u>z/OS UNIX System Services</u> <u>Command Reference</u>.

Unicode Services

z/OS UNIX exploitation of Unicode Services is functionally similar to that provided for Enhanced ASCII. The basic EBCDIC nature of the z/OS platform remains. Likewise, programs cannot alter their EBCDIC nature as compiled units, except for C programs, which can be compiled as ASCII. Locale restrictions that apply to Enhanced ASCII functions apply to Unicode Services functions as well.

For more information about Unicode Services, see z/OS Unicode Services User's Guide and Reference.

File tagging in Unicode Services

Files that are tagged can be converted between any CCSID of the program or user and the CCSID of the file, if Unicode Services supports that conversion. Unlike Enhanced ASCII, which affects conversion of regular file, pipes, and character special files, an environment enabled for Unicode Services environment affects regular files and pipes only. No character special support beyond that provided for Enhanced ASCII is included.

Automatic code set conversion

Automatic conversion of files from one code set to another is controlled globally by the AUTOCVT (ON| OFF) parameter in the BPXPRMxx parmlib member. AUTOCVT can be overridden by individual programs at a thread level, and therefore is a controlling switch only for programs that do not explicitly establish their own conversion options. The default setting for AUTOCVT is OFF.

Although the value of AUTOCVT can be changed using the SETOMVS command, changing the conversion mode does not affect conversion of opened files for which I/O has already started.

When AUTOCVT (ON) is set, every read or write operation for a file is checked to see if conversion is necessary. A performance penalty is therefore involved, even if no conversion occurs. It is recommended that AUTOCVT be left off and each program be enabled for conversion.

For information about commands that allow or disallow automatic code set conversion by default, see Controlling text conversion for z/OS UNIX shell commands in z/OS UNIX System Services Command Reference.

Porting considerations

If your system administrator enabled Enhanced ASCII or Unicode Services, you can able to tell z/OS UNIX which files are ASCII (code set ISO8859-1) files and which files are EBCDIC (code set IBM-1047) files. This enhanced functionality is useful when working with portable XPG 4.2 applications that are written in ASCII. You can port an application to z/OS UNIX, compile it in ASCII, and also tag text files as ASCII. z/OS UNIX performs conversion when an ASCII program reads or writes an EBCDIC tagged file or when an EBCDIC program reads or writes an ASCII tagged file.

For more information about enabling Enhanced ASCII or Unicode Services at the system level, see <u>converting files between code pages</u> and <u>Using Unicode Services</u> in <u>z/OS UNIX System Services Planning</u>. For information about porting applications to z/OS UNIX, see <u>z/OS UNIX System Services Porting Guide</u>.

Chapter 16. Working with directories

This information covers these topics:

- The working directory
- Displaying the name of your working directory
- Changing directories
- Creating a directory
- · Removing a directory
- · Listing directory contents
- · Comparing directory contents
- Finding a directory or file

The working directory

The shell always identifies a particular directory within which you are assumed to be working. This directory is known as the *working directory* (also known as the *current working directory*). To work with a file within your working directory, you need specify only the file name with a command. If you want to work with a file in another directory, you can change your working directory, using the cd shell command and naming the new directory.

Tip: Instead of changing directories, you could use relative notation to access a file in a different directory; see "Using notations for relative path names" on page 192 for more information.

When you type the OMVS command and begin working in the shell environment, you are placed in your *home directory* as your working directory.

Displaying the name of your working directory

To check on the name of the directory you are currently working in, just enter the pwd command (print working directory).

If Alice Smith is working in her home directory, for example, the system displays the name of her working directory in this form:

/u/smitha

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

If Alice Smith enters the command cd projecta, the **projecta** subdirectory of her home directory becomes her working directory. If she issues the pwd command, it displays:

/u/smitha/projecta

Note: A directory name can be specified in two ways, with or without a trailing slash; for example:

/u/smitha/projecta /u/smitha/projecta/

In this topic, a trailing slash is not used.

Changing directories

Use the cd command to change from one working directory to another. If you have permission to access the directory, you can move to any directory in the file system by using cd and the path name for the directory:

cd pathname

See <u>Chapter 18, "Handling security for your files," on page 217</u> for more information on directory permissions.

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

cd

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

cd projectb

To check that she has changed directories, Alice types pwd and the system displays:

/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

If you use the ls -a command to list the contents of a directory, you see that every directory contains the entries . (dot) and .. (dot dot):

. (dot)

This refers to the working directory.

.. (dot dot)

This refers to the parent directory of your working directory, immediately above your working directory in the file system structure.

If one of these is used as the first element in a relative 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 z/OS shell in several forms:

Notation	Meaning				
~	Your home directory (that is, the directory given by your HOME environment variable). The command:				
	cp ~/file1 file2				
	copies file1 in your home directory into file2 in your working directory. This works regardless of what your working directory is.				
	cp file1 ~/dir				
	copies file1 from the working directory into dir in your home directory.				
~ +	The variable \$PWD (which contains the name of your working directory).				
-	The variable \$OLDPWD (which gives the name of the working directory you were in immediately before the last cd command).				
~login name	That user's home directory.				
	Example: To display the profile file of allane , from that user's home directory, issue:				
	cat ~allane/.profile				
	This is useful if there are a group of you working on a project and you have read- write access to some of each other's files.				
	Note: In the z/OS shell, your <i>login name</i> is your TSO/E user ID.				

Example

Suppose that your home directory is /u/turbo and you are working in /u/turbo/prog/src, and you want to display the file limits in the directory /u/turbo/appl/hdr. You could refer to the file in several different ways:

cat ../../appl/hdr/limits
cat ~/appl/hdr/limits
cat /u/turbo/appl/hdr/limits

Creating a directory

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 that is created with mkdir is:

owner=rwx group=rwx other=rwx

For directories, execute permission means permission to search the directory. The octal representation of these permissions is 777 (7 for the owner permission bits, the group permission bits, and the other permission bits).

The new directory, projecta, is one level under her working directory. Figure 15 on page 194 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 15: Creating a new directory

If you want to create a new directory that is not under your working directory, specify an absolute path name. Both directory names and file names can be up to 255 characters long. To distinguish between directory names and file names, you might want to adopt a naming convention.

Your business use naming conventions for directories. For example, a typical convention is for each user to be assigned a directory that is based uniquely on the TSO/E user ID to make the name unique. Only that user would have write access to the directory. For information on how to change access permissions for a directory or file so that other users can read or write to it, see <u>Chapter 18</u>, "Handling security for your files," on page 217.

To create a new directory by using TSO/E, enter:

MKDIR 'directory_name' MODE(directory_permission_bits)

where *directory_name* specifies the path name of the directory to be created. The path name can be a full path name or a relative path name. Specify the name, which can be up to 1023 characters long, in single quotation marks. Specify MODE, the directory permission bits, in three octal characters; they can be separated by commas or blanks. The default mode (read-write-execute permission) is:

```
owner = rwx
group = r-x
other = r-x
```

The octal representation of these permissions is 755. (When MKDIR is used to create a directory, the default permission bits are different from when mkdir is used.) Here execute permission means permission to search the directory.
For example, to specify a directory with a full path name and mode 700, enter:

MKDIR '/u/smitha/umods' MODE(7,0,0)

Use a full path name with the MKDIR command. When a relative path name is specified, MKDIR defines the directory in the user's home directory, regardless of the working directory. If user Alice Smith is in her home directory smitha and wants to create a directory with a relative path name and the default mode, she can enter:

MKDIR 'umods'

The directory umods is one level under her home directory, smitha. Its full path name is /u/smitha/ umods.

Removing a directory

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

rmdir directory

To remove your working directory, you must first move into another working directory.

To delete the files in a directory and the directory itself in one step, use the rm command with the -r option. The format of the command is:

rm -r file

where file is the name of the directory. Be careful! You may want to use the -i option so that you will be prompted to confirm the deletions:

rm -ri file

Listing directory contents

The 1s command lists the contents of a directory. To see the contents of your working directory, enter:

ls

To list the contents of a different directory, add the relative or absolute name of the directory you want to look at, as in:

ls dira/dirb ls abc/def/ghi

1s displays directory contents in alphabetic order. Typical 1s output looks like:

bin csrb.cpy fifotest makefl temp.t cc etc helplist phones.com totals

1s does not normally distinguish between directories, regular files, and special files. If you want a list of directory contents that distinguishes 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:

```
/
Directory
*
Executable file
```

FIFO special file

@

L

Symbolic link

&;

External link

If there is no character following the file name, the file is none of these types.

1s 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 1s uses.

The ls command with the -E option displays a character indicating whether or not the program is loaded from the shared library region. If the program is from the shared library region, an 'l' will appear as the fourth character in the second column. If the program is not from the shared library region, a '-' will appear. For example:

-ps-	1 FRED	SYS1	101	0ct	02	16:30	james
a-s-	1 FRED	SYS1	654	0ct	02	16:30	backup
a	1 FRED	SYS1	40	0ct	02	16:30	temp
ap-l	1 FRED	SYS1	562	0ct	02	16:34	diag
sl	1 FRED	SYS1	106	0ct	02	16:53	bird
	-ps- a-s- a ap-l sl	-ps- 1 FRED a-s- 1 FRED a 1 FRED ap-1 1 FRED s1 1 FRED	-ps- 1 FRED SYS1 a-s- 1 FRED SYS1 a 1 FRED SYS1 ap-1 1 FRED SYS1 s1 1 FRED SYS1	-ps- a-s- 1 FRED SYS1 101 a-s- 1 FRED SYS1 654 a 1 FRED SYS1 40 ap-l 1 FRED SYS1 562 sl 1 FRED SYS1 106	-ps- a-s- 1 FRED SYS1 101 Oct a-s- 1 FRED SYS1 654 Oct a 1 FRED SYS1 40 Oct ap-l 1 FRED SYS1 562 Oct sl 1 FRED SYS1 106 Oct	-ps- a-s- 1 FRED SYS1 101 Oct 02 a-s- 1 FRED SYS1 654 Oct 02 a-s- 1 FRED SYS1 40 Oct 02 ap-1 1 FRED SYS1 562 Oct 02 s1 1 FRED SYS1 106 Oct 02	-ps- 1 FRED SYS1 101 Oct 02 16:30 a-s- 1 FRED SYS1 654 Oct 02 16:30 a 1 FRED SYS1 40 Oct 02 16:30 ap-1 1 FRED SYS1 562 Oct 02 16:34 sl 1 FRED SYS1 106 Oct 02 16:53

In this example, the files *james*, *backup*, and *temp* are not loaded from the shared library region, but the files *diag* and *bird* are.

Comparing directory contents

You can use the command:

diff -r dir1 dir2

to check whole directories for changes. With the -r option, diff compares the files in **dir1** with the files in **dir2** that have the same names.

This command can be useful if you have two directories that hold different versions of the same files and subdirectories.

You can use the -r option with other commands. For example:

cp -r dir1 dir2

copies all the files and subdirectories from dir1 to dir2.

rm -r dir

removes all the files and subdirectories under dir and then removes dir itself.

Finding a directory or file

The find command lists the names of all the files under a directory with a given characteristic or set of characteristics. The simplest version of the command is:

find dirname

It displays the names of all files under the given directory, including files in subdirectories under the directory.

To display the names of all files whose names have the form specified in *pattern*, issue:

find dirname -name pattern

Example: To list the names of all files under the directory abc with the file name extension .lst,issue: (

find abc -name '*.lst'

The asterisk (*) is a wildcard character that stands for any sequence of zero or more characters.l Using find, you can locate files quickly, even when you have a complicated file system structure, with many directories and subdirectories. See the find command description in <u>z/OS UNIX System Services Command</u> Reference.

Chapter 17. Working with files

This information covers these topics:

- Using an editor to create a file
- Naming files
- Deleting a file
- Deleting files over a certain age
- Identifying a file by its inode number
- Creating links
- Deleting links
- Renaming or moving a file or directory
- Comparing files
- Sorting file contents
- Counting lines, words, and bytes in a file
- · Searching files by using pattern matching
- Browsing files
- Simultaneous access to a file
- Backing up and restoring files
- · Listing process IDs of processes with open files

Using an editor to create a file

When you are logged into the shell, you have a choice of editors to use to create and change files, depending on which terminal interface you are using, OMVS or the asynchronous terminal interface. For details about the editors, see Chapter 19, "Editing files," on page 227.

If you are using NFS from your workstation, you can edit z/OS UNIX files directly with your editor of choice.

When you create directories and files, you can control access to them. Whenever you want, you can change the *access permissions* that are set when you first create a directory or file. See <u>Chapter 18</u>, "Handling security for your files," on page 217 for more information about 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.

The POSIX portable file name character set (see <u>"The POSIX portable file name character set" on page</u> <u>306</u>) is a subset of the POSIX portable character set, which is listed in <u>"The POSIX portable character set"</u> on page 306.

The POSIX portable character set (see <u>"The POSIX portable character set" on page 306</u>) is a complete list of all valid characters for a file name.

Restriction: Double-byte characters are not supported in a file name and are treated as single-byte data. Using double-byte characters in a file name might cause problems. For instance, if you use a double-byte 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 shells are case-sensitive, and distinguish characters as either uppercase or lowercase. Therefore, FILE1 is not the same as file1.

A file name can include a suffix, or *extension*, that indicates its file type. An extension consists of a period (.) and several characters. For example, files that are C code could have the extension . c, as in the file name dbmod3.c. Having groups of files with identical suffixes makes it easier to run commands against many files at once.

Processing in uppercase and lowercase

Case-sensitive processing means that an environment distinguishes and handles characters as either uppercase or lowercase: FILE1 is not the same file as file1. The availability of case-sensitive processing depends on the environment:

Shell

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

ISPF

To issue a TSO/E command with a z/OS UNIX path name and get case-sensitive processing of the path name, enter the command on a command line that supports mixed-case processing, for example the Command Processor panel (usually ISPF option 6). Some ISPF option panels convert the command and file name to uppercase before they are processed.

The default ISPF edit profile usually folds to uppercase the data you enter in a file. To prevent this, type caps off on the command line before you begin working in the file. After you enter caps off, it remains in your profile.

If you are working on a file and realize that you have been typing in uppercase when you really wanted lowercase, you can change the contents of the file to all lowercase. Type this on the command line:

c all p'>' p'<'

TSO/E

Case-sensitive. Follow the syntax rules of the command you are using. For instance, make sure to enclose a path name in single quotation marks when using commands such as ALLOCATE, OPUT, and so on.

JCL

Case-sensitive. You can specify z/OS UNIX files in DD statements by giving the absolute path name (no relative path names) and enclosing the names in single quotation marks. Be careful to keep JCL keywords such as DD, PATH, and so on, in uppercase.

Note: Traditional MVS utilities may define their own requirements for allowing mixed-case file names to be specified as input (as compared with the rules for specifying mixed-case file names on DD statements in JCL). For example, you need to use the binder's CASE=MIXED option if you want to bind a load module into the file system and give the load module a lowercase name.

Deleting a file

The command rm can delete, or remove, several files at once. For example:

rm file1 file2 file3

removes all the specified files.

Suppose Alice Smith's directory **projectb** had several old meeting notices in it that she wanted to delete: **0607.mtg**, **0615.mtg**, **0623.mtg**, and **0628.mtg**. She could remove all four with just a single command:

rm 06*.mtg

Tip: Be careful when using the wildcard asterisk (*) for removing files; you may want to use the -i option, which prompts you to verify the deletion.

For the tcsh shell, see <u>"Displaying deletion verification" on page 60</u> for more information on how to control the wildcard asterisk.

Deleting files over a certain age

The skulker shell script provides a way to delete files in a directory based on comparing the file's access time to a specified age. This can be useful for removing temporary files created by utilities, or files that were intended to be temporary but were forgotten about.

The skulker script is a z/OS shell script, and can be easily modified to fit any particular system or user need. The script is located in **/samples**, but the system administrator should have relocated it somewhere else. Check with the system administrator for the location of the script. You should copy the script into your home directory or subdirectory, where you can modify it if you desire different removal criteria.

It is also possible to invoke the skulker script with the cron daemon so that it may be run on a regular basis.

The format for running the skulker script is as follows:

skulker [-iw] [-r|-R] [-l logfile] directory days_old

The -i option displays the files that are candidates for deletion, and then prompts the user to terminate the script or continue with the deletion.

The -w option does not delete the files, but sends a warning to the owner of each file (via mailx) that the file is a candidate for deletion.

The -r option moves recursively through subdirectories, finding non-directory files that are equal to or older than the specified number of days. The -r option is mutually exclusive with the -R option.

The -R option moves recursively through subdirectories, finding both non-directory files and subdirectories that are equal to or older than the specified number of days. Any subdirectories that are found as candidates for deletion are only deleted if they are empty after all their contents (files, subdirectories and files in subdirectories) that are candidates for deletion have been deleted. The -R option is mutually exclusive with the -r option.

The -1 *logfile* specifies a *logfile* to store a list of files that have been deleted, are candidates for deletion, or for which warnings have been mailed; and any errors that might have occurred.

directory specifies the directory in which to look for files that are candidates for deletion.

days_old specifies the age of files you want to remove, based on when the file was last accessed.

For more information about the skulker script, including restrictions, see <u>skulker</u> in <u>z/OS UNIX System</u> Services Command Reference. In addition to its file name, each file in a file system has an identification number, called an *inode number*, that is unique in its file system. The inode number refers to the physical file, the data stored in a particular location. A file also has a device number, and the combination of its inode number and device number is unique throughout all the file systems in the hierarchical file system.

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

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

ls -i

If Alice Smith issues that command for her **proja** directory, she sees the following display:

1077 inspproc 1077 isoproc 1492 kgnproc 1500 mcrproc

Because the files inspproc and isoproc are hard-linked, they have the same inode number.

Creating links

A *link* is a new 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.

You can use the ln command to create a hard link or a symbolic link. A file can have an unlimited number of links to it.

Creating a hard link

A *hard link* is a new name for an existing file. You cannot create a hard link to a directory, and you cannot create a hard link to a file on a different mounted file system.

All the hard link names for a file are of equal importance with its original name. They are all real names for the one original file. To create a hard link to a file, use this command format:

```
ln old new
```

Thus, *new* is the new path name for the existing file *old*. In Figure 16 on page 203, /u/benson/proja is the new path name for the existing file /u/smitha/proja.



Figure 16: Hard link: a new name for an existing file

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 16 on page 203. Because an inode number represents a physical file in a specific file system, you cannot make hard links to other mounted file systems.

Creating a symbolic link

You can create a symbolic link to a file or a directory. Additionally, you can create a symbolic link across mounted file systems, which you cannot do with a hard link. A *symbolic link* is another file that contains the path name for the original file; in essence, a reference to the file. 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:

ln -s old new

Thus, *new* is the name of the new file containing the reference to the file named *old*. In Figure 17 on page 203, /u/benson/proja is the name of the new file that contains the reference to /u/smitha/proja.



Figure 17: Symbolic link: a new file

When you create a symbolic link, you create a new physical file with its own inode number, as shown in Figure 17 on page 203. 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 182.

Creating an external link

An *external link* is a special type of symbolic link, a file that contains the name of an object outside of the z/OS UNIX file system. Using an external link, you associate that object with a path name. For example, setlocale() searches for locale object files in the z/OS UNIX file system, but if you want to keep your locale object files in a partitioned data set, you can create an external link in the file system that points to the PDS. This will improve performance by shortening the search made by setlocale().

A file can be an external link to a sequential data set, a PDS, or a PDS member. When a file is an external link to an MVS data set, an NFS client user can use the path name to access the data set. To use the path name to edit, browse, or display the attributes of the data set that is the target of an external link, you must be using the Network File System feature. Working in a shell, you can create (1n) an external link, display information (1s) about the link (not the target of the link), or delete (rm) the link.

These services support external links:

- NFS client: You can create external links as files within the z/OS UNIX file system and then access these files as an NFS client user to access the MVS data sets that they point to.
- A program using the exec() family of functions, the BPX1EXC (exec) callable service, the BPX1LOD (loadhfs) callable service, or the BPX1SPN (spawn) callable service can access an MVS data set using an external link. This capability includes external link programs that are invoked as commands in the shell.
- Dynamic link libraries: The external link name used on a DLL load is a member name. For example, you would code a link as:



Figure 18: External link: A new file

WS is the member name that is linked to the file wwwss.so.

ln -e IMWYWWS /usr/lpp/internet/bin/wwwss.so

To create an external link to a data set, use this command format:

ln -e old new

In Figure 18 on page 204, /u/brooks/plib/pgm1 is the name of the new file that contains the reference to the partitioned data set BROOKS.PLIB.PGM1.

Limitations of an external link: z/OS UNIX C programs running cannot fopen() or fread() an external link. For more information, see the <u>In</u> command description in <u>z/OS UNIX System Services Command</u> *Reference*.

Due to NFS protocol limitations, -e does not create an external link on NFS. For more information on creating an external link on NFS, see creating an external link in *z/OS Network File System Guide and Reference.*

Deleting links

To delete a file that has hard links, you must enter rm against all the link names, including the original file name. If you try to delete a file that is hard-linked, its contents do not disappear until you remove every link to it.

To delete a file that is a symbolic link, you enter rm against the symbolic link name. This removes the link, not the file it refers to. When you delete a file that is symbolically linked, any remaining symbolic links refer to a file that no longer exists. If you know the names of the symbolic link files, you may want to delete them.

To delete a file that is an external link, run rm against the external link name. If you delete a data set that is externally linked, the remaining external link refers to a data set that no longer exists.

Renaming or moving a file or directory

You can use the mv command to move or rename files. For example:

mv file1 file2

moves the contents of *file1* to *file2* and deletes *file1*. This is similar to:

cp file1 file2 rm file1

except that, when the files are in the same mountable file system, mv renames the file rather than copying it. *file1* and *file2* do not have to be in the same directory.

The mv command can move several files from one place to another.

For example:

mv file1 file2 file3 directoryb

moves all three files to directoryb.

Using the -R or -r option, you can move a directory and all its contents (files, subdirectories, and files in subdirectories) into another directory. For example:

mv -R directorya directoryb

Comparing files

Consider the following situation: A warehouse has an *active file* that keeps track of current inventory. As goods are brought in, appropriate records are added to the file. As orders are shipped out, the records are deleted. At the end of the day, the warehouse makes a copy of the active file to keep as a permanent journal.

It would be useful for such a business to be able to compare one day's journal to another day's to see what has changed. This can be done with the diff command:

diff oldfile newfile

compares the two files. The output of diff shows lines that are in one file but not in the other. The lines in *oldfile* but not in *newfile* are displayed with a < in front of them. Lines in *newfile* but not in *oldfile* are displayed with > in front.

For example, say you have a file wmnhist.text with one line in it:

Susan B. Anthony awoke one morning

Then you created a copy of the file with the command:

cp wmnhist.txt newhist.txt

You use an editor—either the ISPF editor or the ed text editor—to change the first line in **newhist.text** to:

Sojourner Truth awoke one morning

You save the file. Now you enter the command:

diff wmnhist.txt newhist.txt

diff displays:

1c1 < Susan B. Anthony awoke one morning ---> Sojourner Truth awoke one morning

The 1c1 at the beginning of the diff output indicates that line 1 in the old file has changed (c) when compared with line 1 in the new file. diff shows what must be changed in the first file to make it look like the second file. Remember this sequence when you look at the output of diff. Here the first file, **wmnhist.txt**, contained the line Susan B. Anthony awoke one morning where the second file, **newhist.txt**, has Sojourner Truth awoke one morning.

New lines are indicated with an a (add lines), and lines that should be deleted are indicated with a d (delete). See the <u>diff</u> command description in <u>z/OS UNIX System Services Command Reference</u> for more details.

diff helps you determine what has changed in the time that elapsed between saving the two files. The same sort of operation is useful in many record-keeping situations, any time you have two different versions of the same file and you want to check the differences.

Sorting file contents

When you create a file of records, you usually do not type the information in any particular order. However, you may want to keep lists in some useful order after you have entered the information. To sort the records in a file, use the sort command. sort assumes two things:

- Your file contains one record per line. To put it another way, there is a single <newline> character between a record and the next record.
- The fields in a record are separated by recognizable characters. In the sample file **comics.lst** in / **samples** (shown in Figure 19 on page 207), we use colons.

Detective Comics:572:Mar:1987:\$1.75 Demon:2:Feb:1987:\$1.00 Ex-Mutants:1:Sep:1986:\$2.60 Justice League of America:259:Feb:1987:\$1.00 Boris the Bear:1:Sep:1986:\$1.50 Flaming Carrot:14:0ct: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

Figure 19: A sample file: comics.lst

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

sort /samples/comics.lst

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

sort /samples/comics.lst >filename

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

sort /samples/comics.lst >sorted.lst

sorts the file and stores the result in **sorted.lst** without changing the input file.

When you use >filename to redirect sorted output into a file, you may want to make the output file name different from the (unsorted) input file name. If you want to overwrite a file with its sorted contents, see the description of the -o flag in the <u>sort</u> command description in <u>z/OS UNIX System Services Command</u> Reference.

Using sorting keys — an example

By default, sort sorts according to all the information in the record, in the order given in the record. Since the name of the comic book is the first thing on the line, the output is sorted according to comic book name. But suppose that you want to sort according to some different piece of information. For example, suppose you want to sort by date of publication. You can do this by specifying sorting keys.

A *sorting key* tells *sort* to look at specific fields in a record, instead of looking at each record as a whole. A sorting key also tells what kind of information is stored in a particular field (for example, an ordinary word, a number, or a month) and how that information should be sorted (in ascending or descending order).

A sorting key can refer to one or more fields. Fields are specified by number. The first field in a record is field number 1, the field after the first separator character is field number 2, and so on. In the comic book list, the month is field number 3, and the year is field number 4.

A single sort command can have several sorting keys. The most important sorting key is given first; less important sorting keys follow. Let us look at an example that sorts by year and then by month within a year. Therefore, the first sorting key refers to the year field, and the second to the month field. To specify a sorting key, use the -k option. This option has the following format:

```
-k start_field[.char1] [opts] [,end_field[.char2] [opts]]
```

where *start_field*, *end_field*, *char1*, and *char2* are all integers.

- start_field indicates which field in the input record contains the start of the sorting key.
- *char1* indicates which character in that field is the first character of the key. Omitting *char1* means the key begins with the first character of the starting field.

In our example, the first sorting key (referring to the year) has a *start_field* value of 4 (since the year is field 4). We do not need to specify *char1*, since we want to start the key with the first character of the year field.

The options, *opts*, are specified with letters; they identify the type of data in the specified field and tell how to sort it. Some of the possible options and their meanings are:

d

Indicates that the field contains uppercase, lowercase, or mixed-case letters, letters and digits, or digits. sort sorts the field in *dictionary* order, ignoring all other characters.

Μ

Indicates that the field contains the name of a month. sort looks only at the first three characters of the name, so January, JAN, and jan are all equal.

n

Indicates that the field contains an integer (positive or negative).

Putting an r after any of these letters tells sort to sort in reverse order (from highest to lowest rather than lowest to highest). For example, Mr means to sort in the order December, November, October, and so on.

In our example the sorting key based on the year uses n. Thus, the sorting key for the year field (4) in the file **comics.lst** is:

-k 4n

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

-k 3M

A sort command that uses sorting keys needs to know which character separates the record fields. You can specify this with the option -t followed by the separator character. The example uses -t:. Therefore, the full sort command is:

sort -t: -k4n -k3M comics.lst >sorted.lst

The file to be sorted comes after the various options. This is the order that you must use. The redirection construct can come anywhere on the line, but is usually put at the end.

Counting lines, words, and bytes in a file

The wc command tells you how big a text document is.

wc file file ...

tells you the number of lines, words, and bytes in each file.

If you want to find out how many files are in a directory, enter:

ls | wc

This pipes the output of 1s through wc. Because 1s prints one name per line when its output is being piped or redirected, the number of lines is the number of files and directories under your working directory.

Searching files by using pattern matching

One of the most common record-keeping operations is obtaining a sublist of a list. For example, you might want to list all the *Watchmen* comics that appear in the main comics list. The command to do this is grep.

The simplest form of the grep command is:

grep word file

where *word* is a particular sequence of characters that you want to find, and *file* is your list of records. grep lists every line in the file that contains the given word. For example:

grep Watchmen comics.lst

lists every line in **comics.lst** that contains the word Watchmen. As another example:

grep 1986 comics.lst

lists every line in **comics.lst** that contains the sequence of characters 1986. Presumably, this lists all the comics that were published in 1986.

grep Jul:1986 comics.lst

lists all the comics published in July 1986.

If the string of characters you want to search for contains a blank, put single 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

The examples of grep, so far, have displayed the records in a file that contain the desired string anywhere in the line. If you want to be more specific—say to find records that *begin* with a certain string of characters (instead of having that string anywhere in the line)—use grep with *patterns* instead of strings.

To understand patterns, it helps to think about the special wildcard characters discussed in <u>"Using a</u> wildcard character to specify file names" on page 74. 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 *pattern characters*, or *metacharacters*. Some pattern characters are:

^ (caret)

Stands for the beginning of a line. For example, ^abc is a pattern that represents *abc* at the beginning of a line.

\$ (dollar sign)

Stands for the end of a line. For example, xyz\$ is a pattern that represents xyz at the end of a line.

. (dot or period)

Stands for any (single) character. For example, a.c is a pattern that represents *a*, followed by any character, followed by *c*.

* (asterisk)

Indicates zero or more repetitions of part of a pattern. For example, .* indicates zero or more repetitions of . (period). Since the . stands for any character, .* stands for any number of characters. For example, $^a.*z$ is a pattern that represents *a* at the beginning of a line, *z* at the end, and any number of characters in between.

A typical grep command has the form:

grep 'pattern' file

This displays all the records in the file that match the given pattern. For example:

grep '^Superman' comics.lst

displays all the records that begin with the word Superman.

grep '00\$' comics.lst

displays all the records that end in 00.

If you want to use the *literal* meaning of a pattern character instead of its special meaning, put a backslash $(\)$ in front of the character.

Example: To find all the lines that end in \$1.00, issue:

grep '\\$1\.00\$' comics.lst

Without a backslash in front of the \$ and . (period), these characters would have their special pattern meanings.

Regular expressions

More complex patterns than the ones discussed here are accepted. The formal name for a pattern is a *regular expression*. For further information, see <u>Regular expressions (regexp)</u> in <u>z/OS UNIX System</u> Services Command Reference.

Browsing files

When you display, or browse, a file, you cannot make any changes to the file while you are viewing it. You can browse a z/OS UNIX file using ISPF or using shell commands. With shell commands, you have the choice of browsing the file in an unformatted or formatted display.

Browsing files without formatting

Using the shell: The z/OS shell has a quick way to find out what is in a given file: the head command and the tail command.

head *filename*

Displays the first 10 lines of the given file or files.

tail filename

Displays the last 10 lines of the given file or files.

Suppose you have a file that contains records sorted according to date. tail tells you the date of the last records in the file, giving you an idea of how current the file's contents are. In a sorted comic book list, for example, tail could show the most recent comics that had been recorded in the file.

To display the contents of an entire file, you can use any of these commands: cat, pg, more, or obrowse.

Using ISPF: To use ISPF to browse a z/OS UNIX file, you can take one of the following actions:

- Enter the TSO/E OBROWSE command followed by the path name for the file. This command displays the file, which you can begin browsing.
- Select an option for browse on the ISPF menu, if such an option is available.

After the file is displayed, you can use function keys to scroll forward and backward in the file.

For complete information about browsing, see *z/OS ISPF User's Guide Vol II*.

Browsing files with formatting

Using the shell: The term *formatting* refers to 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 could be using ed to edit the same file at the same time. When a file has been accessed by more than one user simultaneously, the last changes saved overwrite any previous changes.

In a program, you can use byte-range locking to avoid this problem. For more information about byte-range locking in a program, see *z/OS XL C/C++ Programming Guide*.

You can use the Network File System feature to coordinate locking of remote files and directories. See <u>"Using the Network File System feature" on page 186</u> for an overview of this feature. For more detailed information, consult the appropriate Network File System documentation.

Backing up and restoring files: options

There are several options for backing up and restoring files:

- Data Facility System-Managed Storage Hierarchical Storage Manager (DFSMShsm) provides automatic backup facilities for data sets. The systems programmer uses DFSMShsm facilities to back up mountable file systems by backing up the data sets that contain them on a regular basis; the data sets can be restored when necessary. DFSMShsm is also used for migrating (archiving) and restoring unmounted file systems.
- Tivoli[®] Storage Manager (TSM), formerly known as ADSTAR Distributed Storage Manager (ADSM), provides a backup function for z/OS UNIX clients. There are two types of backup: *incremental*, in which all new or changed files are backed up; and *selective*, in which the user backs up specific files.

Backup can be performed automatically or when the user requests it. The user can initiate a specific type of backup or start the scheduler, which will run whatever action the administrator has scheduled for the user's machine.

• From the shells, you can manually back up data by using the TSO/E OGET command to copy files into an MVS sequential data set, partitioned data set, or partitioned data set extended (PDSE) that you know is backed up. To simplify archiving multiple files, the **pax** or **tar** utilities can be used to consolidate

individual component files into a single archive file that can then be copied to an MVS data set. **pax** and **tar** can write the archive directly to an MVS data set, eliminating the need to copy the archive manually with OGET. For more information about using **pax** or **tar** and OGET to backup and restore file from the shell, see "Backing up and restoring files from the shell" on page 212.

You can use the **cron** utility to automatically start running **pax** or **tar** commands at a specified time.

After the files are in an MVS data set, you can load the data set to a tape. Conversely, you can load files from a tape into an MVS data set and then copy them into the file system. For more information, refer to "Transporting an archive file on tape or diskette" on page 270.

Backing up and restoring files from the shell

This information describes how to use the pax or tar utilities to back up and restore files. The purpose of both utilities is to store the data and attributes of one or more *component files* into a single file, referred to as the *archive file*. pax is considered to be the standard utility for managing archive files, replacing tar; therefore, pax is used as the default utility in the examples that follow. However, tar is still widely used, and in the z/OS environment provides practically equivalent function. Therefore, the corresponding tar commands are also shown.

Both pax and tar support multiple archive formats and options that allow a greater or lesser degree of file characteristics to be preserved. The USTAR format allows the most information to be saved, therefore it is used as the default format in the examples that follow. For more information about the USTAR and other archive formats, refer to <u>z/OS UNIX System Services Command Reference</u>. Because both pax and tar can read and write archives in USTAR format, either utility can be used to restore an archive that was created by the other. The significant difference between the two utilities is that only pax can perform code page conversion on files during creation of, or extraction from, an archive. Users of tar can use the iconv utility to perform the same conversion on files as a separate step.

Both pax and tar support inline compression and decompression of files. Because compressed archives occupy an average of 50-60% percent of the uncompressed archive, many of the examples shown here use compression. Note that compressed archives are not guaranteed to be portable to other UNIX systems.

Archives can be copied to an MVS data set using the TSO/E OGET command and later copied back to the file system using the TSO/E OPUT command. For OS/390[®] Release 8 and later, pax and tar can read and write archives that reside in an MVS data set, making it unnecessary to first manually move files between the file system and MVS using OGET or OPUT.

pax and tar support file names and link names that exceed 100 characters in length. The utilities remain compatible with other UNIX systems and with previous versions of OS/390.

The remainder of this topic describes the following specific steps for backing up and restoring files to and from an MVS data set and performing other related archive management tasks.

- Backing up a complete directory into an MVS data set
- Restoring a complete directory from an MVS data set
- · Viewing the contents of an archive
- Converting between code pages
- Appending to an existing archive
- · Storing selected files into an archive
- Restoring selected files from an archive
- · Appending to an existing archive
- Backing up selected files by date

These examples demonstrate the most common tasks related to backing up and restoring files, and do not attempt to describe all of the options of the pax and tar utilities. See <u>z/OS UNIX System Services</u> <u>Command Reference</u> for a complete description of pax and tar.

Backing up a complete directory into an MVS data set

To back up the complete directory **/u/project**, including the subdirectories and their contents, into a compressed archive stored in the MVS data set 'PROJECT.ARCHIVE', enter the following commands:

```
cd /u/project
pax -wzvf /tmp/project.pax.Z ./
tso "oget '/tmp/project.pax.Z' 'PROJECT.ARCHIVE' binary"
```

Note:

1. The pax command can write directly to the MVS data set; you can skip the OGET command by specifying the MVS data set on the pax command:

pax -wzvf "//'PROJECT.ARCHIVE'" ./

2. The equivalent tar commands are:

tar -cUzvf /tmp/project.pax.Z ./

To write directly to MVS (OS/390 Release 8 or later):

tar -cUzvf "//'PROJECT.ARCHIVE'" ./

- 3. You change to the current directory first in order to simplify the pax/tar command, and so that the files are stored in the archive using a path name that is relative to the current directory. This simplifies the task of restoring the archive later to a different directory. The "./" is used rather than an asterisk to collect any component files that begin with "." in the current directory.
- 4. The archive is written to a directory that is not in the source path that is being archived, in order to prevent pax/tar from trying to store the archive within itself. Doing so can cause pax/tar to loop infinitely during creation, and can result in corrupted files during restore.
- 5. Naming archives with a suffix of "pax.Z" (or "tar.Z") is not required by pax/tar, but is done as a convention to identify them as pax or tar archive files. The ".Z" is used to identify a compressed file.
- 6. The -z option is used to turn on compression, and is not required.
- 7. The -v option is used to display the names of files as they are being stored, and is not required.

Restoring a complete directory from an MVS data set

To restore the directory backed up in the previous example to **/u/project_old**, enter the following commands:

```
tso "oput 'PROJECT.ARCHIVE' '/tmp/project.pax.Z binary"
cd /u/project_old
pax -pe -rvf /tmp/project.pax.Z
```

Note:

1. The pax command can read an archive directly from an MVS data set; you can skip the OPUT command by specifying the MVS data set on the pax command:

pax -pe -rvf "//'PROJECT.ARCHIVE'"

2. The equivalent tar command is:

tar -p -xvf /tmp/project.pax.Z

To read directly from MVS (OS/390 Release 8 or later):

tar -p -xvf "//'PROJECT.ARCHIVE'"

3. The -pe option for pax and the -p option for tar are used to restore the original owner, group, modes, and extended attributes. If you do not have the appropriate privileges to restore these, warning

messages are generated. These options are not required to restore the component files and can be omitted. For tar, the -o option is also used to disable restoring the owner and group.

- 4. pax and tar automatically detect the archive format and whether the archive is compressed, so the z option for pax and, for tar, the -U option is not required. If these options are used, pax/tar fails if the archive is not compressed or not in USTAR format.
- 5. The -v (verbose) option is used to display the names of files as they are being restored, and is not required.
- 6. Component files can be renamed during extraction by pax using the -i or -s option.

Viewing the contents of an archive

To view the contents of the **/tmp/project.pax.Z** archive created in the previous step, enter one of the following commands:

To list only the names of component files:

pax -f /tmp/project.pax.Z

To list the contents in a verbose format similar to "1s -1":

pax -vf /tmp/project.pax.Z

For OS/390 Release 7 and later, to list the extended attributes in a verbose format similar to "1s -E":

pax -Ef /tmp/project.pax

Note: The equivalent tar commands are:

- To list only component files: tar -tf /tmp/project.pax.Z
- For a verbose list: tar -tvf /tmp/project.pax.Z
- For extended attributes (OS/390 Release 7 or later):

tar -tEf /tmp/project.pax.Z

Converting between code pages

Archives are often used to move files between UNIX systems. When an archive contains text files, it is frequently the case that the file must be converted from the source system's default code page to the target system's code page. You can do this by using the **iconv** utility on each file before storing it in an archive or after restoring it from an archive. The pax utility, however, provides an inline code page translation option, -o that can simplify this task. For example:

• To convert component files from EBCDIC (IBM-1047) to ASCII (ISO8859-1) when storing them in an archive:

pax -o to=iso8859-1 -wzvf /tmp/project.pax.Z ./

 To convert component files from ASCII (ISO8859-1) to EBCDIC (IBM-1047) when extracting them from an archive:

pax -o from=iso8859-1 -pe -rzvf /tmp/project.pax.Z

Note:

- 1. The -o option allows both a "from" and a "to" code page to be specified on the same command. If a "from" or "to" codepage is not specified, pax assumes it to be EBCDIC (IBM-1047).
- 2. For more information about the code sets supported for this command, see the Coded Character Set Conversion Table in *z/OS C/C++ Programming Guide*.

Converting archives that contain text and non-text component files. Archives often contain both text and non-text files. Examples of non-text files are image files, such as JPGs and GIFs, and other pax/tar

archives. When the -o option is specified, pax converts all files, regardless of type. This corrupts non-text files. The general approach for overcoming this limitation is to run pax two or more times against the same archive, extracting component files in groups of text and non-text types. Whether it is easier to identify (by file name) text files or non-text files will determine how you approach this.

For example, suppose you wish to restore the archive **mywebsite.pax**, which consists of HTML files (text files) and JPG files (JPEGS, non-text image files) and was created on a system whose default code page is ASCII (ISO8859-1), into the directory **/u/website**. Assume that the majority of the files are HTML files and that the archived files represent several levels of subdirectories.

First, restore the entire archive using the -o option:

pax -rvf mywebsite.pax -o to=IBM-1047

This extracts and converts all component files. The extracted non-text JPEG files would be corrupted because they were also converted. The next step would be to re-extract the JPG files without the -o option. The pax option allows you to specify a "pattern" that will be used to extract only those files that match the pattern. However, because of the multiple subdirectories, there is no way to create a pattern that would match every JPG in each subdirectory. Instead, a list of file names to be extracted must first be created and then used as the pattern for the pax command to extract the files. Issuing the following command in the z/OS shell would accomplish this:

pax -rvf mywebsite \$(pax -f mywebsite.pax | grep -i JPG\$)

This command consists of two parts:

pax -rvf mywebsite \$()

and

```
pax -f mywebsite.pax | grep -i JPG$
```

The first part is simply the regular pax command for extracting files from an archive. The \$() expression says to first run the command between the parentheses and substitute the results in place. The second part is the command that generates a list of file names in the archive that end in "JPG" (or any mixed-case variation).

The previous example shows one approach. In general, for any archive, the breakdown of text to non-text files and the uniqueness of the names that identify each type dictate the manner and order in which the files are extracted. For example, we could have reversed the process by first extracting all files without using the -o option, and then re-extracting the HTML files on the second command using the -o option to convert the files

Appending to an existing archive

To add additional files and directories to a previously created *uncompressed* archive, use the **-a** (append) option.

Example: To add the file oops.forgot to the existing archive allfiles.pax, issue: :

pax -awvf allfiles.pax oops.forgot

Result: The file **oops.forgot** is added to the end of the archive. If a file with the same name already exists in the archive, it will not be overwritten or replaced.

Note:

- 1. You can append directly to archives in sequential MVS data sets only. **pax** and tar do not support appending to archives that reside in partitioned MVS data sets.
- 2. The equivalent tar command is:

```
tar -rvf allfiles.pax oops.forgot
```

Backing up selected files by date

The following examples pertain to the z/OS shell only, and demonstrate how to back up selected files that may have been modified within a specified number of days. To do this you create a "find" command that returns the list of files that meet the specified criteria, and then use the output from this command as the list of files input to pax.

Example: To back up all files in the directory /u/source that have been modified in the last week, issue:

pax -wzvf backup.pax.Z \$(find /u/source -type f -mtime -8)

Example: To back up all files in the directory **/u/usrtools/** that have not been accessed in the last 100 days, issue:

```
pax -wzvf backup.pax.Z $( find /u/usrtools -type f -atime +100 )
```

Note: The tar equivalent for the pax portion of the previous commands is:

tar -czUvf backup.pax.Z

Listing process IDs of processes with open files

It is often helpful to know which processes have open files. This information can be provided with the fuser utility.

The fuser utility lists the process IDs of all processes on the local system that have one or more named files open.

The syntax of the command is as follows:

fuser [-cfku] file

file is the path name of the file for which information is to be returned, or, if the-c option is used, the path name of a file on the file system for which information is to be reported.

Option

Description

-с

Reports on all open files within the file system of which the specified file is a member.

-f

Reports on only the named files. This is the default for this command.

-k

Sends the SIGKILL signal to each local process. Note that only a superuser can terminate a process that belongs to another user.

-u

The user name associated with each process ID is written to standard error.

Chapter 18. Handling security for your files

Each user has user ID (UID) and group ID (GID) numbers that are set when the user is defined to the system. A user always belongs to at least one group—for example, a department—and each group that uses the system is assigned a GID. The system uses the UID and GID to identify the files and processes that a user may use. When you create a directory or a file, it is automatically associated with your UID, and its GID is set to the owning GID for the *parent directory* (the directory it is in).

There are three classes of users whose access you can control with the permission bits (ACLs allow access control for any user or group):

- Owner (the owner of the file or directory whose UID matches the UID for the file)
- · Group (a member of the group whose GID matches the GID for the file)
- Other (anyone else)

You control access to a file and directory *that you own* through its permission bits. (Taken together, the permission bits are often called the *mode*.)

In this topic, we discuss:

- Default permissions set by the system
- · Changing permissions for files and directories
- Using the sticky bit on a directory to control file access
- Auditing file access
- Displaying file and directory permissions
- · Setting the file mode creation mask for programs
- · Changing the owner ID or group ID associated with a file
- Temporarily changing the user ID or group ID during execution
- Displaying extended attributes
- · Using access control lists (ACLs) to control access to files and directories

Default permissions set by the system

When you first create a file or directory, the system sets default *read*, *write*, and *execute* (*rwx*) permissions. The meanings of the three permissions differ somewhat for a file and a directory:

Permission	Notation	Meaning
read	r	Directory: Permission to read, but not search, contents.
		File: Permission to read or print contents. To run a shell script, you need both read and execute permission.
write	W	Directory: Permission to change the directory, adding or deleting members.
		File: Permission to change the file, adding or deleting data
execute	x	Directory: Permission to search a directory. Usually r and x are used together.
		File: Permission to run a file—that is, enter it as a command. Typically this permission is used for shell scripts and for files containing executable programs. (To run a shell script, you need read and execute permission.)

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

Using	To create a	Default permissions
mkdir shell command	Directory	owner=rwx group=rwx other=rwx
		In octal form: 777
MKDIR TSO command	Directory	owner=rwx group=r-x other=r-x In octal form: 755
ICL with po BATHMODE	Directory or file	
specified	Directory of file	owner= group= other=
		In octal form: 000
ISPF editor, OEDIT command, oedit command	File	owner=rwx group= other=
		In octal form: 700
vieditor	File	owner=rw- group=rw- other=rw-
		In octal form: 666
ed editor	File	owner=rw- group=rw- other=rw-
		In octal form: 666
Redirection (>)	File	owner=rw- group=rw- other=rw-
		In octal form: 666
cp command	File	Sets the output file permissions to the input file permissions.
OCOPY command	File	Permission bits for a new file are specified with the ALLOCATE command, using the PATHMODE keyword, prior to entering the OCOPY command. If the PATHMODE keyword is omitted, the default is:
		owner= group= other=
		In octal form: 000

Using	To create a	Default permissions
OPUT or OPUTX	File	For a text file:
command		owner=rw- group= other=
		In octal form: 600
		For a binary file:
		owner=rwx group= other=
		In octal form: 700

For more information on octal numbers, see "Using octal numbers to specify permissions" on page 220.

Changing permissions for files and directories

You can use the chmod command to set or change permissions for your files and directories. To change permissions, you must be the owner or a superuser. (If you are uncertain about ownership, use the ls -l command and look for your TSO/E user ID.)

You can specify the chmod command like this:

chmod mode pathname

You can specify the mode in symbolic form or as an octal value. For more information on the chmod command, see the chmod command description in *z/OS UNIX System Services Command Reference*.

Using a symbolic mode to specify permissions

A symbolic mode has the form:

[who] op permission [op permission ...]

The *who* value is optional; it can be any combination of the following:

u

Sets owner (user) permissions.

g

Sets group permissions.

0

Sets other permissions.

а

Sets all permissions; this is the default.

The *op* part of a symbolic mode is an operator that tells chmod to turn the permissions on or off. The possible values are:

+

Turns on a permission.

Turns off a permission.

=

Turns on the specified permissions and turns off all others.

To set the permission part of a symbolic mode, you can specify any combination of the following permissions in any order:

r

Read permission.

s

This stands for set-user-ID-on-execution or set-group-ID-on-execution permission. See <u>"Temporarily</u> changing the user ID or group ID during execution" on page 225 for more information.

t

This sets the sticky bit on, for a file or directory.

Directory: The sticky bit is set on for a directory so that a user cannot remove or rename a file in the directory unless one or more of these conditions is true:

- The user owns the file.
- The user owns the directory.
- The user has superuser authority.

File: The sticky bit is set for frequently used programs in the file system, to reduce I/O and improve performance. When the bit is set on, z/OS UNIX searches for the program in the user's STEPLIB, the link pack area, or the link list concatenation. For information on copying a load module from the file system into a data set, see <u>"Copying an executable module from the file system" on page 265</u>. See <u>Verifying that the sticky bit is on in z/OS UNIX System Services Planning</u> for information on using the sticky bit with daemons.

W

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

х

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

Χ

Search permission for a directory; or execute permission for a file only when the current mode has at least one of the execute bits set.

For example, to turn on read, write, and execute permissions, and turn off the set-user-ID and sticky bit attributes for a file, enter the command:

chmod a=rwx file

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

Using octal numbers to specify permissions

Typically, octal permissions are specified with three or four numbers, in these positions:

1234

Each position indicates a different type of access:

- In position 1 are the bits that set permission for set-user-ID on access, set-group-ID on access, or the *sticky bit*. Specifying this position is optional.
- In position 2 are the bits that set permissions for the owner of the file. Specifying this position is required.
- In position 3 are the bits that set permissions for the group that the owner belongs to. Specifying this position is required.
- In position 4 are the bits that set permissions for others. Specifying this position is required.

Position 1

Specifying the bits in position 1 is optional. For position 1, you can specify these octal numbers:

0

1

Off

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

No access (---)

Specifying these bits is required. For each type of access—owner, group, and other—there is a corresponding octal number:

0

```
1
```

Execute-only access (--x)

2

Write-only access (-w-)

3

Write and execute access (-wx)

4

Read-only access (r--)

5

Read and execute access (r - x)

6

Read and write access (rw-)

7

Read, write, and execute access (rwx)

To specify permissions for a file or directory, you use at least a *three-digit* octal number, omitting the digit in the first position. When you specify three digits instead of four, the first digit describes owner permissions, the second digit describes group permissions, and the third digit describes permissions for all others.

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

Table 11: Three-digit permissions specified in octal					
Octal number	Permission	Meaning			
666	6 6 6 / \ IW- IW- IW-	owner (rw-) group (rw-) other (rw-)			

Table 11: Three-digit permissions specified in octal (continued)				
Octal number	Permission	Meaning		
700	7 0 0 / \ Iwx	owner (rwx) group () other ()		
755	7 5 5 / \ IWX I-X I-X	owner (rwx) group (r-x) other (r-x)		
777	7 7 7 / \ IWX IWX IWX	owner (rwx) group (rwx) other (rwx)		

Using the sticky bit on a directory to control file access

Using the mkdir, MKDIR, or chmod command, you can set the sticky bit on a directory to control permission to remove or rename files or subdirectories in the directory. When the bit is set, a user can remove or rename a file or remove a subdirectory only if one of these is true:

- The user owns the file or subdirectory.
- The user owns the directory.
- The user has superuser authority.

If you use the rmdir, rename, rm, or mv utility to work with a file, and you receive a message that you are attempting an operation not permitted, check to see if the sticky bit is set for the directory the file resides in.

Auditing file access

Using the chaudit command, you can specify which types of file access are audited by RACF. RACF writes the audit information to system management facilities (SMF) record 80.

Only a file owner or a security auditor can specify if auditing is turned on or off, and when audit records should be written for a directory or a file: for successful accesses, failed accesses, or for all accesses.

You can specify audits for read, write, and search or execute attempts. For each of these, you can specify audits for successful access, failed access, or both. You can also set the audit flags off, so that audits are not performed.

The default audit bits are set at file creation:

- The user-requested-audit flags are set to audit failed attempts to read, write, or execute. Only the file owner or a superuser can specify user audit options.
- The auditor-requested-audit flags are set off (no auditing). To specify auditor audit options, you must have security auditor authority.

See the <u>chaudit</u> command description in <u>z/OS UNIX System Services Command Reference</u> for a description of the chaudit command. See the topic about <u>specifying file audit options</u> in <u>z/OS UNIX System Services</u> Planning for a description of how a superuser or security auditor would use the chaudit command.

Displaying file and directory permissions

To display the permissions for the files and directories in your working directory, use ls -W. (The ls -l command displays all the access permissions but does not display the audit permissions.) The display format is:

drwxr-x	fff	2	ELVIS	64MB	96	Jun	15	10:34	statrp
-rwx	fff	1	ELVIS	64MB	107	Jul	10	07:45	jun93
- IWX	fff	1	ELVIS	64MB	80	Aug	09	13:15	jul93
-rwx	fff	1	ELVIS	64MB	150	Sep	15	10:45	aug93
drwxr-xr-x	fff	2	ELVIS	64MB	96	Jun	17	09:05	dbappl
-rwxr-x	fff	1	ELVIS	64MB	150	Jun	17	10:15	txn1

• First field: A string of 10 characters. The first character indicates the file type. The next 9 characters are the permissions. For example:

-rwxr-xr-x

View them this way:

- IWX I-X I-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
 - 1 for a symbolic link
 - p for a named pipe (FIFO special file)

In the example, - indicates a regular file.

- The first set of 3 characters show the owner's permissions. In this example, the owner has read, write, and execute permission (rwx).
- The second set of 3 characters show the group permissions. In this example, the group to which the user belongs has read and execute permission (r-x).
- The third set of 3 characters show the other permissions. In this example, any other user can read the file and execute it (r-x). If the sticky bit is on, you see a T or t in the final field (--T or --t).
- Second field: The audit settings. These 6 characters are actually two groups of 3 characters. The first group of 3 describes the audit settings requested by a user; the second group describes audit settings requested by a security auditor. The characters can be:
 - s to audit successful access attempts
 - f to audit failed access attempts
 - a to audit all accesses
 - for no audit

In the example, fff---,

fff means *failed* read, write, and execute or search attempts to access the file are audited by the user.

--- means read, write, and execute or search attempts to access the file are not audited by the security auditor.

- Third field: The number of links to the file or directory.
- Fourth field: The owner's login name (TSO/E user ID).

Note: When files owned by user ID 0 (UID=0) are transferred from any UNIX-type system across an NFS connection to another UNIX-type system, the user ID changes to -2 (UID=-2). On a z/OS UNIX system, -2 is not a valid user ID; therefore, 1s displays UID 4294967294 (the unsigned equivalent of -2).

- Fifth field: The name of the group associated with the file or directory.
- Sixth field: The size of the file, expressed in bytes.
- Seventh field: A date and time. For a file, this is the time the file was last changed; for a directory, it is the last time a file was created or deleted in the directory.
- **Eighth field:** The name of the file or directory. If the file is a symbolic link, that also is indicated. See the additional information for the filename **lnk** in this example:

l----- 1 ELVIS SYS1 8 May 21 15:30 lnk -> /tmp/ehk \$

Setting the file mode creation mask

When a file is created, it is assigned initial access permissions. If you want to control the permissions that a program can set when it creates a file or directory, you can set a *file mode creation mask* using the umask command.

You can set this file mode creation mask for one shell session by entering the umask command interactively, or you can make the umask command part of your login. When you set the mask, you are setting limits on allowable permissions: You are implicitly specifying which permissions are *not* to be set, even though the calling program may allow those permissions. When a file or directory is created, the permissions set by the program are adjusted by the umask value: The final permissions set are the program's permissions minus what the umask values restrict.

To use the umask command for a single session, enter:

umask *mode*

and specify the mode in either of the formats used by chmod: symbolic (rwx) or octal values. The symbolic form expresses what can be set, what is *allowed*, while octal values express what cannot be set, what is *disallowed*. For example, both of these commands set the same umask:

```
umask a=rx
umask 222
```

To display the mask,

- If you just enter umask, you see the mode displayed in octal values, indicating what *cannot* be set.
- If you enter umask -S, you see the mode displayed in symbolic form, indicating what can be set.

The shell's initial setting of the mask is 000, which means that read, write, and execute permission can be set on for everyone. But the system-wide profiles provided with the product set the mask to 022.

Changing the owner ID or group ID associated with a file

The user might need to change the UID or GID for a file. To protect the data in a file from unauthorized users, the system controls who can change the file access:

- To change the owner (UID) of a file, the superuser can enter a chown command.
- To change the group (GID) of a file, the superuser or the file owner can enter a chgrp command, specifying either a RACF group name or a GID. The file owner must have the new group as his group or one of his supplementary groups.

Superuser tasks are discussed in <u>Using the BPX.SUPERUSER resource in the FACILITY class</u> in <u>z/OS UNIX</u> System Services Planning.

Temporarily changing the user ID or group ID during execution

An executable file can have an additional attribute, which is displayed in the execute position (x) when you issue 1s -1. This permission setting is used to allow a program temporary access to files that are not normally accessible to other users. An s or S can appear in the execute permission position; this permission bit sets the effective user ID or group ID of the user process that is executing a program to that of the file whenever the file is run. The setuid and setgid bits are only honored for executable files that contain load modules.

s

In the owner permissions section, s indicates that the set-user-ID (S_ISUID) bit is set and execute (search) permission is set.

In the group permissions section, s indicates that the set-group-ID (S_ISGID) bit is set and execute (search) permission is set.

S

In the owner permissions section, S indicates that the set-user-ID (S_ISUID) bit is set, but the execute (search) bit is not.

In the group permissions section, S indicates that the set-group-ID (S_ISGID) bit is set, but the execute (search) bit is not.

A good example of this behavior is the mailx utility. A user who is 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 action. The mail program does.

Displaying extended attributes

The -E option on the 1s shell command displays extended attributes. For more information about this option, refer to <u>"Executable modules in the file system" on page 180</u>.

Using access control lists (ACLs) to control access to files and directories

Using access control lists (ACLs), you can control access to UNIX files and directories by individual users (UIDs) and groups (GIDs). ACLs are used in conjunction with permission bits.

There are three kinds of ACLs:

- Access ACLs are ACLs that are used to provide protection for a file system object.
- *File default ACLs* are model ACLs that are inherited by files created within the parent directory. The file inherits the model ACL as its access ACL. Directories also inherit the file default ACL as their file default ACL.
- *Directory default ACLs* are model ACLs that are inherited by subdirectories created within the parent directory. The directory inherits the model ACL as its directory default ACL and as its access ACL.

There are two kinds of ACL entries:

- *Base ACL entries* are permission bits (owner, group, other). You can change the permissions using chmod or setfacl.
- *Extended ACL entries* are ACL entries for individual users or groups. Like the permission bits, they are stored with the file, not in RACF profiles.

Additional access control mechanisms are allowed to further restrict the access permissions that are defined by the file permission bits. Because ACLs can grant and restrict access, the use of ACLs is not UNIX 95-compliant.

ACLs are supported by HFS, zFS, and TFS. It is possible that other physical file systems will eventually support z/OS ACLs. Consult your file system documentation to see if ACLs are supported.

Setting up ACL support

Using access control lists (ACLs) in *z/OS UNIX System Services Planning* provides detailed information on setting up and managing ACLs. It also explains the considerations involved when you are using ACLs in a sysplex and how ACLs are used in file access checks. To add, delete, or update an ACL, or update the permission bits, use the setfacl shell command. The getfacl shell command displays the contents of an ACL. The ls with -l output will also indicate if extended ACL entries exist.

See <u>ACL tasks and their associated commands</u> in *z/OS UNIX System Services Planning* for a chart that shows how various shell commands are used when working with ACLs. For complete information on the commands involved, see *z/OS UNIX System Services Command Reference*.

Chapter 19. Editing files

When you are logged into the shell, you have a choice of editors to use to create and change files, depending on which interface you are using:

• OMVS terminal interface:

- The full-screen ISPF editor, which you can invoke using the OEDIT or oedit command.
- The ed editor, a line editor
- The sed stream editor, a noninteractive editor. It is intended for systematic editing; you invoke the
 editor with a file of editing commands and a target data file and it produces an edited target file, with
 no user interaction.

• Asynchronous terminal interface:

- The vieditor, an interactive editor

If you are using NFS from your workstation, you can directly edit z/OS UNIX files with your workstation editor of choice.

Using ISPF to edit a z/OS UNIX file

ISPF Edit provides a full-screen editor you can use to create and edit z/OS UNIX files. You can access ISPF Edit in several ways:

- Using the oedit shell command
- Using the TSO/E OEDIT command at the TSO/E READY prompt or from the shell command line
- From the ISPF menu (if a menu option is installed)
- From the ISPF shell (accessed using the TSO/E ISHELL command)

Tip: If you know you will be using OEDIT or OBROWSE during a shell session, make your initial invocation of the shell from ISPF. If you enter the OMVS command from ISPF, you can subsequently access OEDIT and OBROWSE more quickly than if you had entered the OMVS command from TSO/E.

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

When you are working in MVS (TSO/E or ISPF), your home directory is the default working directory.

When you create a new file, you must have the appropriate permissions to add a new file to the parent directory. When a file is created using ISPF Edit, its default permissions are:

```
owner = rwx
group = - - -
other = - - -
```

The octal number is 700.

ISPF Edit allows only one edit session at a time per file. It reads the entire file when the edit session begins. At the end of the session, it replaces the original file with the edited file.

During an ISPF Edit session, you can use these types of commands:

Type of commands	Usage notes
Scrolling commands	You can use commands to scroll the data up, down, left, or right.

Type of commands	Usage notes
Line commands	You perform line editing by entering a <i>line command</i> directly on the line number of the affected line. For example, to delete a line, you enter D on the line number; to repeat a line, you enter R on the line number. You can enter line commands for several lines at the same time.
Primary commands	To perform general editing tasks, you enter <i>primary commands</i> at the command line on the panel. For example, you can use the FIND command to scan data for a specific character string. If you entered:
	FIND printf(
	on the command line, your cursor moves to the next occurrence of printf(. Likewise, you can enter the CHANGE command to make global changes within a file.
	Example: To change all instances of CRTL to C-RTL, issue:
	CHANGE CRTL C-RTL ALL
External data commands	While you are editing one file, you can use external data commands to work with another file, a sequential data set, or a member of a partitioned data set or PDSE—moving data to or from the file you are editing. ISPF Edit provides five external data commands: COPY, MOVE, REPLACE, CREATE, and EDIT.

To end an edit session:

- Saving all changes, enter the END command or press <F3>.
- Without saving any changes, enter the CANCEL command.

When you end the edit session, you go back to where you were when you began it: on the entry panel, on an ISPF command line, at the TSO/E READY prompt, or at the shell prompt.

All you ever wanted to know about ISPF Edit

The discussion in this topic is an introduction to ISPF Edit. For detailed information about ISPF Edit, use the online help facility or refer to *z/OS ISPF Edit and Edit Macros*.

Using the vi screen editor

The vi editor is available if you login to the shell using rlogin or telnet. It is not available if you login using the OMVS command. The vi editor is a full-featured text editor with the following major features:

- Full-screen editing and scrolling capability
- · Separate text entry and edit modes
- Global substitution and complex editing commands using the underlying ex commands.

This overview just introduces some fundamentals to help you get started. For more information, see <u>Appendix A</u>, "Advanced vi topics," on page 275 and the <u>vi</u> command description in <u>z/OS UNIX System</u> Services Command Reference.

Basic principles

To begin using vi, you type the command:

vi filename

where *filename* is the name of a file you want to edit. This can be an existing file, or it can be a new file that you want to create.

The vi command begins a vi session. In a vi session, you enter input that creates or changes the contents of the file specified on the command line. vi reads and uses the input you type until you quit your vi session.

In a vi session, you are always in one of two modes:

- **Insert Mode**, in which everything you type is taken as text input. vi displays text on the screen as you enter it. Eventually, vi stores this text in a file.
- **Command Mode**, in which vi interprets everything you type as a command to change the text in some way. Usually, commands do not appear on the screen—you just see the effects of the command. For example, if you enter the command to delete a line of text, you see the line disappear, but you never see the delete line instruction that you actually typed.

To switch from Insert Mode to Command Mode, simply press the key marked <Esc>. If you are not sure which mode you are in, press <Esc> several times. This always brings you back to Command Mode.

To delete a character, you must be in Insert Mode. Pressing <Backspace> deletes the last character you typed; pressing <Backspace> twice deletes the last two characters, and so on. vi usually does not immediately delete these characters on the screen—it just backs up the cursor so that anything you enter is typed over the characters that were there. When you leave Insert Mode, vi adjusts the screen to remove any characters that were deleted by <Backspace> and not over-typed.

To quit a vi session, do one of these:

- :wq to save your changes and quit vi
- :q! to quit without saving your work

A simple vi session

This information shows you how to edit a simple text file. Try it to get the feel of using vi. You can edit the text file:

vi1.txt

which is supplied as part of the z/OS shell. It is in the directory /samples. To do this, copy this file to current working directory:

cp /samples/vi1.txt vitest

Now, begin your vi session by typing:

vi vitest

vi clears the screen, then displays the contents of the file., vi also displays:

"vitest" 30 lines, 668 characters

This tells the name of the file being edited and how big it is.

The cursor is positioned at the beginning of the file. Use the keys listed in <u>Table 12 on page 229</u> to position the cursor anywhere on any line in the file:

Table 12: vi editor: Positioning the cursor				
To move the cursor:	Press			
Down a line	j or↓(the Down arrow key)			
Up a line	k or ↑ (the Up arrow key)			
Left along a line	h or ← (the Left arrow key)			

Table 12: vi editor: Positioning the cursor (continued)	
To move the cursor:	Press
Right along a line	l or → (the Right arrow key)

Note: The arrow keys do not work on all terminals.

To experiment a bit more, move the cursor to the beginning of the first line in the file, then press 5 followed by \rightarrow . You do not see the 5 displayed anywhere—but when you press \rightarrow , you see the cursor move five characters to the right. As a general rule, when you type a number followed by an action, vi repeats the action that number of times.

By the way, ask yourself if you are in Insert Mode or Command Mode. You must be in Command Mode because the characters you type (for example, the 5) do not appear on the screen. When you start a vi session, you always begin in Command Mode.

Adding text

The simplest action you can perform is adding text to what is already on the screen. Move the cursor to the blank line following:

And frightened Miss Muffet away.

The cursor should be at the first position in the blank line. Now type a. Because you are in Command Mode, this is taken to be a command, not text. The a command tells vi to begin adding to the text that is already on the screen. If you now type:

Little Boy Blue

you can see the characters appear on the line. The a command switches from Command Mode to Insert Mode. You can now see what you are typing.

Press <Enter> at the end of the line. The bottom part of the screen moves down to make a new blank line after the line you were typing. Keep typing more lines:

Come blow your horn The sheep's in the meadow, The cow's in the corn.

You see that the bottom part of the screen keeps moving down to make more room for what you are typing. After the a command, the text that you type is added into the middle of existing text.

When you have typed the last line, press <Enter> to make a new blank line, then press <Esc>. <Esc>switches from Insert Mode back to Command Mode. Now, vi interprets what you type as commands again. If you type 4 followed by ↑, the cursor moves up four lines to the beginning of the text you just typed in. The 4 does not appear on the screen when you type it, because command input is not usually displayed.

Move the cursor to the B at the beginning of the word Blue in the text you have just typed. Press a to add more text, then type the letter 1. The 1 is added after the B and the rest of the text on the line moves over to make room for the new character. This shows that a adds text after the current cursor position.

Press <Backspace>. The cursor backs up one space. Press <Esc>to return to Command Mode. The 1 disappears when you leave Insert Mode, and vi adjusts the screen to get rid of characters deleted by backspacing.

The Little Boy Blue rhyme that you have just added to the file follows the previous nursery rhyme immediately. The file would look better with a blank line separating the two rhymes. Figure out how to put in this blank line, and do it.
Moving the cursor up and down the screen

You already know how to move the cursor up and down; however, this can be a slow process if you have a large file that you want to move through quickly. To speed this process up, vi offers several commands that can jump the cursor up or down many lines at a time.

In Command Mode, use the following commands:

Command

Moves the cursor:

Η

To the upper left hand corner of the screen. H stands for High and it moves the cursor as high on the screen as it can go.

L

To the bottom of the screen. L (uppercase) stands for Low.

Μ

To the middle of the screen. M stands for Middle. Experiment with these commands to see how they move the cursor.

Moving up and down through a file

While you are editing a file, you can move through it one line at a time, several lines at a time, or screens at a time. You can use these commands to move up and down through a file:

Command

Moves the cursor:

<Ctrl-D>

Down (or forward) half a screen. The cursor stays where it is -- the text moves underneath it.

<Ctrl-F>

Down (or forward) almost a full screen. This lets you move forward through the file very rapidly.

<Ctrl-U>

Up (or backwards) half a screen.

<Ctrl-B>

Up (or backwards) almost a full screen.

If you move forward far enough through **vitest**, you will see a number of lines that are blank except for a tilde (~) as the first character. These lines are actually beyond the end of the file -- the file ends with the line:

And the mome raths outgrabe.

vi could just show an empty screen after this last line, but then you would not know if the screen was empty because you had reached the end of the file or if the file just contained a lot of blank lines; therefore, vi uses ~ to mark lines that are past the end of the file.

Moving the cursor on the line

You can also move the cursor by whole word boundaries, using word-motion commands. Make sure that you are in Command Mode (press <Esc>). 0 and \$ let you move back and forth on a line quickly.

Command

Moves the cursor:

[^] or **0**

To the beginning of the current line (to the first nonblank space). The command 0 is short for 0|, which moves the cursor to column number 0.

\$

To the end of the current line

\$ stands for the end of the line in a number of vi commands.

Go to the beginning of a line, and press w. The cursor jumps forward to the beginning of the next word on the line. w stands for word and it moves the cursor forward one word. If you keep pressing w, the cursor keeps jumping forward. When you jump forward from the last word in the line, you go to the first word in the next line. If you precede w with a number (as in 5w), the cursor jumps forward that many words.

Typing b is like typing w, except that you go back a word instead of forward. If you go back from the first word on a line, you get to the last word on the previous line. If you precede b with a number (as in 3b), the cursor jumps backward that many words.

If the cursor is in the middle of a word, typing e moves the cursor to the end of the word. For example, if the cursor is in the middle of the word slithy, typing e moves the cursor to the last letter in the word. If the cursor is already on the last letter of a word, typing e moves the cursor to the end of the next word.

To move the cursor between words *including punctuation* (that is, punctuation is considered to be a word), use the following commands:

Command

Moves the cursor:

е

To the end of the current word

W

To the beginning of the next word

b

To the beginning of the previous word

To move the cursor between words *ignoring punctuation* (that is, punctuation is skipped), use the following commands:

Command

Moves the cursor:

Е

To the end of the current word

W

To the beginning of the next word

В

To the beginning of the previous word

Moving to sentences and paragraphs

To move between sentences and paragraphs, use the following commands:

Command

Moves the cursor:

```
)
```

To the beginning of the next sentence

(

To the beginning of the preceding sentence

}

To the beginning of the next paragraph

{

To the beginning of the preceding paragraph

These commands can also be preceded by a number to change the effect of the command. For example, 3) moves the cursor forward 3 sentences.

Deleting text

There are several commands that delete text from the screen. All of these begin with the letter d. After the d comes a letter indicating what you want to delete. Usually this letter is based on one of the cursor movement commands. For example:

Command

Action

d\$

Deletes text from the cursor's current position to the end of the line.

dd

Deletes the entire line containing the cursor.

dL

Deletes text from the cursor's current position to the bottom of the screen.

dw

Deletes text from the cursor's current position to the beginning of the next word.

de

Deletes text from the cursor's current position to the end of a word. If the cursor is in the middle of a word, de deletes to the end of the same word; if the cursor is at the end of a word, de deletes to the end of the next word.

In the same way, d followed by \rightarrow or \leftarrow (1 or h) can delete a single character. Try both instructions and see which character gets deleted.

If you delete something by accident, you can undo the deletion by typing u (lowercase). Try this now. Type dH. What happens? Now type u and see the deleted text return.

A number followed by a delete command repeats the command that number of times. For example:

- 5dw deletes five words
- 10dd deletes ten lines

Changing text

To change existing text, use the c command the same way you use d. c is a combination of d and a—it deletes text, then begins to append text to replace what was deleted.

Command

Action

c\$

Lets you change everything from the cursor's current position to the end of the line.

сL

Lets you change everything to the end of the page.

сс

Lets you change all of the current line, regardless of the cursor position.

Go to the beginning of the first line of **vitest** and type c\$. vi puts a \$ at the end of the line. The \$ marks the end of the block of text that vi intends to change. If you now begin typing something like The rain in Spain, you type over the text that was previously on the line. If you keep typing, you eventually type over the \$. The \$ was never there -- it was just a marker to show the block of text to be replaced.

After a c command, the text you type shows up on the screen. This means that c puts you in Insert Mode. When you finish typing replacement text, you must press <Esc> to return to Command Mode.

You can enter any amount of text to replace existing text. For example, c\$ only gets rid of part of a line, but you can enter many lines of replacement text.

Undoing a command

If you make a change and then realize it was in error, you might still be able to correct it.

Command Action

u

Undoes the last command entered.

U

Undoes all changes made to the current line.

Saving a file

When you finish editing text, you must save your work in a file. Until you save your work, your text is on the screen but it is not recorded in any usable way. When you quit vi, your work disappears unless it is saved.

If you started your vi session with vi filename, it is easy to write the edited text back into the same file. In Command Mode, just type:

:w

and press <Enter>. When you type the colon, it appears at the very bottom of the screen. The w also appears . When you press <Enter>, there is a short pause and then vi displays some statistics about the saved text: the name of the file, and the number of lines and characters saved.

If you want to save your changes and quit vi, enter:

:wq

If you want to save your text in a different file, type:

:w newfilename

and press <Enter>. Again, this appears . After you save your work, you can quit vi by typing:

:q

Normally, vi does not let you quit before saving; if you do, you lose everything you have done since the last time you saved. If you really want to quit vi without saving your work, type:

:q!

If the file system that you are attempting to save your file to is full, you will see the following message:

FSUM7971 Write error (out of space?)

At this point, you should issue a command to save your file to a new file system where space is available. This can be done by typing:

:w newfilesystem/newfilename

where *newfilesystem* is the name of another file system that has space available, and *newfilename* is the name you wish to call the file.

Once the original file system has space available, you can safely copy the file back to that location.

Searching for strings

In a large document, searching for a particular text string can be very time consuming. The / command prompts for a string to search for in the file. When you press <Enter>, vi searches the file for the next occurrence of the string you entered.

To try searching for a string, first move to the top of **vitest**. Then type:

/Blu

and press <Enter>.

As soon as you enter /, it is displayed on the bottom of the screen. As you type the string Blu, it is echoed . You can use <Backspace> to fix mistakes as you type the search string. After you press <Enter>, the cursor moves to the first occurrence of the string.

The n command searches for the next occurrence of the last string you searched for. Try it now by entering:

n

The cursor should move to the next occurrence of the string, which is the th in the word with. You can also use N like n to search the other direction through the file.

If you just type a slash without anything after it, vi looks for the most recent word or phrase you searched for.

Searching backwards through a file

To specify a search string for a backward search through the file, use the ? command in the same way as /. If you just type a ? without anything after it, vi searches backwards for the most recent word or phrase you searched for. When you search backwards, the n command moves the cursor backward to the next occurrence of the string, and the N command moves the cursor forward.

Case-sensitive searching

When you type in characters after a slash or question mark, make sure you enter them in the correct case. For example, ask vi to search for IN, and type the word in uppercase. You will see that vi prints the message Pattern not found. As it turns out, this file does not contain the word IN in uppercase, although it has the word several times in lowercase.

Notice that the message used the word Pattern. In a vi command, anything after a slash or question mark is called a *pattern*.

Special search characters

In order to make searching more useful, vi gives special meanings to several characters when they are used in patterns. For example, the circumflex or caret character (^) stands for the beginning of a line. Move the cursor to the next line and type:

/^All

vi will look for the word All occurring at the beginning of a line.

The end of a line is represented by the dollar sign (\$). Move the cursor to the next line and type:

/plum\$

You will see that vi searches forward for a line that ends in the word plum.

Inside patterns, the dot (.) stands for any character. For example, move the cursor to the top of the file and type:

/t.e

You will see that the cursor moves to the word the. Type / over and over, and you will see the cursor keep jumping forward to any sequence of three letters that starts with t and ends in e. Were you surprised that the cursor jumped into the middle of the word slithey? vi finds character strings, even when they are in the middle of larger words.

Inside patterns, a dot followed by an asterisk (.*) stands for any sequence of zero or more characters. For example, type:

/^A.*g\$

You will find the next line that begins with the letter A, ends with the letter g and has any number of characters in between.

Character Stands for:

Stands I

Beginning of the line

\$

۸

End of the line

Any character

.*

Any sequence of zero or more characters

vi gives special meanings to several other characters inside patterns. For complete details, see <u>Regular</u> expressions (regexp) in <u>z/OS UNIX System Services Command Reference</u>. A regular expression is the POSIX name for a pattern; here we use the word *pattern* because it is more descriptive.

What happens if you want to search for a character that has a special meaning in patterns? For example, suppose you want to search for the string 2.3*25 somewhere in a file. If you just type:

/2.3*25

vi will think the 3* stands for zero or more occurrences of the digit 3, not the * character. In such cases, put a backslash (\) in front of any characters with special meanings, as in the example:

/2\.3*25

Notice that we had to put a backslash in front of the dot as well as the asterisk; both have a special meaning in patterns.

By default, all searches in vi wrap around from the bottom of the file to the top. Similarly, if you use question marks to search backward through a file, the search will wrap around from the top of the file to the bottom, if necessary.

Moving text

The first step in moving a block of text is to select text for moving. In fact, you already know how to do this. The d command not only deletes a block of text, but also copies it to a paste buffer. Once in the paste buffer, the text can be moved by repositioning the cursor and then using the p command to place the text after the current cursor position.

To delete the first line of the file, move there and type:

dd

The line is deleted and copied into the paste buffer, and the cursor is moved to the next line in the file. To paste the line following the current line, type:

р

To paste text before the cursor rather than after it, use the P (uppercase) command.

If you delete a letter or word size block, it will be pasted into the new position within the current line. For example, to move the word came after the word spider, you could use the following command sequence:

```
/came <Enter>
dw
/spider <Enter>
p
```

Copying text

You copy text in the same manner as you move it, except that instead of using the delete text command d, you use the yank text command, y. The y command copies the specified text into the paste buffer without deleting it from the text. It follows the same syntax as the d command. You can also use the shortcut yy to copy an entire text line into the paste buffer, in the same way as dd.

For example, you can copy the first two lines of the file to a position immediately underneath them. To do so, enter the following command sequence from the first line of the file:

2уу ј р

Note that you must move down one line using j, or the two lines will be pasted after the first line rather than after the second.

Other vi features

Here are a few more helpful vi subcommands:

J

Joins the following line to the current line

Repeats the last command

S

Substitutes the current character with the following entered text **x**

Deletes the current character

Message: vi/ex edited file recovered

Have you received mail with this subject: "vi/ex edited file recovered"? This is what the mail messages look like:

vi -r NoFilename

This message is being sent to you because the exrecover command recovered text files from working files created by ex or vi. When ex or vi is invoked, it first creates these working files in a temporary directory so that it can recover the file being edited if any system errors occur or if the editor is otherwise terminated abnormally.

When vi is invoked, it first creates files in **/tmp** so that it can recover the file being edited if any system errors occur. When vi is invoked from OMVS, it creates its recovery files in **/tmp** but cannot continue.

The current default directory for temporary vi files (usually /tmp) may be implemented as a TFS. In this case, all vi's temporary files that the exrecover daemon uses for recovery would be gone after a system crash. The environment variable TMP_VI can contain a directory path name that can be specified by an administrator as an alternative location for these temporary files. See <u>"Using the TMP_VI environment</u> variable" on page 238 for more information.

The exrecover command automatically recovers these files. By default, this command is started from the **/etc/rc** file. In **/etc/rc** you will see these lines:

Invoke vi recovery
mkdir -m 777 /etc/recover
/usr/lib/exrecover

Every IPL, the **/etc/rc** script is run and the exrecover command is also run. exrecover goes through all the recovery files that were left by the ex or vi editors. These files have names that begin with **VI**; three of them are created for each vi command. exrecover creates directories in **/etc/recover** for each userid, puts the recovered files there, and sends the user mail telling what it did.

Using the TMP_VI environment variable

An administrator can set the TMP_VI environment variable to the path name of an alternate location where vi is to create its recovery files.

Guideline: This environment variable should be set by a system administrator rather than a user. If a user sets the TMP_VI directory to something other than the name that exrecover recognizes as TMP_VI, the user must manually run the exrecover daemon to allow the directory files to be converted to the recoverable files that are used by vi (located in /etc/recover/\$LOGNAME).

Restriction: A system administrator should *not* set TMP_VI to **/etc/recover/\$LOGNAME** or to any directory where a path name component is an environment variable with a user's value that is different from the value of the init process—for example, **\$HOME**.

The temporary **vi** files are converted into a form that is recoverable by **vi** when exrecover is run during IPL. Because exrecover is issued during IPL, it is owned by the init process and, therefore, contains different values for certain environment variables if those environment variables have been set. Throughout the file system, there may be some temporary files that can only be converted by exrecover. This conversion can be done manually by a system administrator to recover files owned by all users or by individual users to recover their own files.

Stopping the mail messages

If no one at your installation intends to use vi, a superuser can get rid of the exrecover mail messages as follows:

- 1. Edit /etc/rc
- 2. Comment out the line that says /usr/lib/exrecover. This stops the exrecover command from running, so no new mail messages will be sent.
- 3.cd /tmp
- 4.rm VI*

If your installation has some users who will be editing with vi, then it's a little trickier. In this case, your vi users will want the recovery capabilities of vi, so you do not want to remove the exrecover command from **/etc/rc**.

Anyone can remove those **/tmp/VI*** files that were generated when users on dumb terminals tried vi. To stop exrecover from sending new mail messages about those files:

- 1. Broadcast a message to make sure no one is using vi at the moment
- 2.cd /tmp
- 3.rm VI*

Deleting the old mail messages

If you want to delete only the mail messages sent by exrecover:

1. Enter

mailx

- 2. Use the mailx commands to read each message: Enter the number of the message
- 3. Enter

d

to delete that message.

To delete all your mail messages, issue:

1.rm /usr/mail/\$LOGNAME

But be careful because this will delete all your mail messages.

Using the ed editor

Using the shell: The ed editor is a line editing program available in the shell for editing text files. When you edit a file with ed, 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*/OS UNIX System Services Command Reference.

Creating and saving a text file

1. To begin editing a new file, enter:

ed filename

where *filename* is the name of a new file.

2. After you see the ?filename message, enter:

а

This indicates that 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 have 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 that were copied—for example, 746. This number includes blanks and newline characters appended to each line of text, which you cannot see on the screen.

If you want to write to a file different from the original *filename*, specify a different *filename* when you enter the w subcommand; for example:

w diffname

Entering the w subcommand does not change the contents of the buffer.

6. To exit the ed program, enter:

q

This deletes the contents of the buffer.

Editing an existing file

To begin editing an existing file, enter:

ed filename

Your current working line is the last line in the file. If you want to change your position in the file before you begin editing, see "Identifying line numbers and changing your position in the buffer" on page 240.

If you are already using ed, have finished editing one file and saved it with the w subcommand, and you now want to edit another file, enter:

e filename

This erases the previous contents of the buffer and loads in the new file.

Identifying line numbers and changing your position in the buffer

To find out how many lines there are in a file, enter:

\$=

To identify the line number of your current working line, enter:

.=

You can make a different line in the file your current working line and then identify its number.

To move the current working line forward a line at a time, press <Enter>. The text of the line is displayed.

To move the current working line backward a line at a time, enter:

-

(hyphen). The text of the line is displayed.

Changing position using numbers

To change the current working line to a different line in the file, enter:

n

where *n* is the number of the line you want to work with. The text of the line is displayed.

To move the current working line *n* lines forward, enter:

.+n

To move the current working line *n* lines backward, enter:

.-n

Changing position using a search string (regular expression)

If you don't know the number or position of the line you want to make your current working line, you can locate a string (or *regular expression*) in the line. To search forward for one or more words or a string of characters, enter:

/regexp/

where *regexp* is one or more words or a string of characters. The line containing the search string is displayed and it is now your current working line.

To search backward for one or more words or a string of characters, enter:

?regexp?

where *regexp* is one or more words or a string of characters. The line containing the search string is displayed and it is now your current working line.

Appending one file to another

If you want to append a file at the end of the file you are working on in the buffer, enter:

r filename

Or, if you want to read a file in after a specific line in the buffer, enter:

nr filename

where *n* is the number of the line in the file.

To display the contents of a file in the edit buffer, enter:

,p,

On your screen, each line of the file is displayed, for example:

```
,p,
Oh, you better watch out
You better not shout
You better not cry
I'm telling you why
```

Once you know the line numbers, you could insert the file **scrooge** after the line You better not cry. Thus, you would enter:

3r scrooge

Displaying the current line in the edit buffer

When you enter subcommands, you identify the current working line with the symbol . (dot).

To display the current working line, enter:

р

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

Use the s (substitute) subcommand and these two special substitution characters to insert text at the beginning or end of a line:

```
^ (circumflex)
```

Inserts text at the beginning of a line

```
$ (dollar sign)
```

Inserts text at the end of a 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:

ns/\$/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 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

Use the t (transfer) subcommand to copy one or more lines within the edit buffer.

To copy one line, enter:

a1tn

where:

a1 is the number of the line to be copied.

n is the number of the line that the line is to be copied after.

To copy a block of lines, enter:

a1,a2tn

where:

a1 is the number of the first line in the block of lines to be copied.

a2 is the number of the last line in the block of lines to be copied.

n is the number of the line that the lines are to be copied after.

To copy lines to the top of the edit buffer, use 0 as the line number for the lines to be copied after.

To copy lines to the bottom of the edit buffer, use \$ as the line number for the lines to be copied after.

Moving lines of text

Use the m (move) subcommand to move a block of lines to a different position in the edit buffer. After the text is moved, the last line in the block of lines becomes the current working line. Enter:

a1,a2mn

where *a1* is the number of the first line in the block, *a2* is the number of the last line in the block, and *n* is the number of the line that the block of lines are to be moved *after*.

To move text to the top of the buffer, use 0 as the line number for the lines to be moved after.

To move text to the end of the buffer, use \$ as the line number for the lines to be moved after.

Undoing a change

To undo a change, use the u subcommand. This subcommand undoes the changes made by the last subcommand that changed the buffer. For the purposes of u, subcommands that change the buffer are: a, c, d, g, G, i, j, m, r, s, t, v, V, and n.

Entering a shell command while using ed

To temporarily switch out of the ed program and run a shell command, enter:

!command name

Ending an ed edit session

When you have finished working with a file, you save the changes by entering:

W

To end the edit session, enter:

q

If you enter q without entering w to first save the buffer, the changes you have made are not saved.

Default permissions

When you create a file using the ed editor, its default permissions are:

owner=rwgroup=rwother=rw-

The octal number is 666.

Using sed to edit a z/OS UNIX file

Using the shell: Because sed is a *noninteractive* editor, you do not use it in an interactive session. Instead, enter the sed command specifying a file that contains 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 that contains 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 the <u>sed</u> command description in <u>z/OS UNIX System Services Command</u> Reference.

Chapter 20. Printing files

If you are a workstation user, you are probably accustomed to having a printer close by, if not on, your desk. In contrast, the MVS system intentionally screens the user from printer knowledge and uses a printer resource pool. One facility provided to manage this pool is the System Display and Search Facility (SDSF).

You can, of course, download z/OS UNIX files and print them at your workstation. However, it may be more convenient to have print jobs sent to accessible Job Entry Subsystem (JES) printers directly by the shell. In addition, you may want to use the large-volume printing facilities offered by MVS.

Formatting files for online browsing or printing

Using the shell: You can use shell commands to format a file for browsing or printing, and then later use the 1p command to send the formatted file to a printer.

If you want to format and print a file immediately, you can request this printing as a single piped command.

To format a z/OS UNIX 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**, and send the output to standard output (your workstation screen). The file appears on your screen in the format you selected. There are many format options for the pr command; see the <u>pr</u> command description in <u>z/OS</u> UNIX System Services Command Reference.

If, instead, you redirect standard output to a file named **report2**, you can later print the file by entering:

lp report2

This requests 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 <u>"Using a</u> pipe" on page 69 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** and pipes the formatted output to tee. tee writes the formatted output to **report2** and at the same time pipes **report2** to the next command, 1p, 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 involved in porting 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 1p command to send a previously formatted file to a JES printer:

lp filename

You can specify more than one file name with the command. The lp command uses existing JES printer facilities. Because a default printer destination is assigned to you, you do not need to specify a destination (with the -d *dest* option) when entering the lp command. However, you can specify a destination other than the default by using the -d *dest* option. For -d *dest*, you can specify LOCAL for any printer or any of the symbolic destination names your systems programmer defined for JES printers. These symbolic names are defined locally.

Class is a frequently used option, and at your site there might be several different classes defined. For instance, C may be designated the class for confidential information. Suppose that you want to print the file **temp.prt** using the default printer destination and specifying class C; you would enter it in either of these ways:

```
lp -d ,c temp.prt
lp -d,c temp.prt
```

The parameters on the -d option are positional, so if you omit a destination, you must still include the comma.

To specify the number of copies you want printed, use the -n option. For example,

lp -n 2 report2

requests the printing of two copies of the formatted file in report2 to the default printer destination.

If you have z/OS Print Server installed on your system, you will use the Print Server version of the 1p command.

Printing with TSO/E commands

Using TSO/E: Some printer services, such as printing a single file to multiple destinations, are not available through the 1p command. To print in TSO/E, you need to know:

- The TSO/E commands you can use to submit print jobs
- · The printing options (class) you want to specify

Here are the steps:

- 1. If you are working in the shell, switch to TSO/E command mode by pressing the TSO function key.
- If you want to print an MVS data set, skip to the next step. If you want to print a z/OS UNIX file, you
 must first copy it into an MVS data set using the TSO/E OGET or OCOPY command. (See <u>"Copying a
 z/OS UNIX file into a sequential data set or PDS member" on page 257</u> for more information on
 copying.)

Tip: Someone at your installation may have written an MVS command list (CLIST) or a REXX program that you can enter as a TSO/E command for printing. The command list could include the OGET or OCOPY command, and would let you specify such things as multiple destinations, special character sets, and notification for a set of people.

- 3. You can format an MVS data set for printing using TSO/E commands. Possibly you will be using ISPF panels.
- 4. Print the data set:

• To enter the request to print the formatted data set, for example, you might enter:

```
printds da(project1.list) class(c)
```

• To submit a print request to the MVS job queue, for example, you might enter:

```
submit jcl.cntl(print1)
```

For a print batch job request, the system returns a message confirming that the job request has been received.

Checking the status of print jobs

If you submit a print job with a shell command, there is no way to check on the status of the job. (The lpstat and cancel commands are not supported.) All output looks the same on the queue in terms of job number. Print jobs could have different setups such as destination or class, but normally the only difference is the number of lines, bytes, or pages and the time of day the output was available to print.

Note: The Print Server is included with z/OS. The Print Server, if enabled, replaces the 1p command and provides other commands, including 1pstat and cancel.

If your operating system includes SDSF, you can use the SDSF panels to monitor and control a TSO/E or batch print job, look at its output as it is running, check its completion, and release it to print.

For a batch job, the STATUS command can provide status if you specify the job name as your user ID followed by one character (for example, MACNEILA). You cannot use the STATUS command for print jobs that you ran using 1p or PRINTDS. STATUS takes either no operands or one or more job names as operands. If you use no operands, the system looks for jobs with names that start with your user ID followed by one character. If you list a job name, it looks for that job name.

Requirement: If you use SDSF to view the output from a job where the job name was assigned using the _BPX_JOBNAME environment variable, you must set the SDSF group function APPC to ON. If APPC is set OFF, the assigned job name will not be displayed, and the jobs will differ only by job number. For more information, see <u>z/OS SDSF Operation and Customization</u>.

Chapter 21. Copying data between the z/OS UNIX file system and MVS data sets

As shown in Figure 20 on page 251, you can copy data between the z/OS UNIX file system and MVS data sets using z/OS UNIX shell commands cp and mv or the TSO/E commands OPUT, OPUTX, OGET, OGETX, and OCOPY.



Figure 20: Copying data between z/OS UNIX and MVS

Copying data using z/OS shell commands

You can use the z/OS shell commands **cp** (copy) and **mv** (move) to copy or move files:

- Between the z/OS UNIX file system and MVS data sets
- Within the z/OS UNIX file system.

With the **cp** and **mv** commands you can specify whether the file or data set is to be copied or moved as text, binary, or as an executable. You can also append or truncate suffixes. For examples of using these commands, see:

- "Using cp to copy a sequential data or PDS member into a z/OS UNIX file" on page 253
- "Using cp to copy a PDS to a z/OS UNIX directory" on page 256
- "Using cp to copy a z/OS UNIX file into a sequential data set or PDS member" on page 258
- "Using cp to copy z/OS UNIX files into a PDS or PDSE" on page 261
- "Copying files within the z/OS UNIX file system" on page 262
- "Using cp to copy executables between MVS and z/OS UNIX" on page 264

For more information about the **cp** and **mv** shell commands, see <u>z/OS UNIX System Services Command</u> <u>Reference</u>.

Copying data using TSO/E commands

You use TSO/E commands to copy data:

- Between the z/OS UNIX file system and MVS data sets
- Within MVS data sets.

The TSO/E commands that enable you to do this are:

OPUT

Puts (copies) an MVS sequential data set or partitioned data set (PDS or PDSE) member into the file system. You can specify text or binary data, and select code page conversion for single-byte data.

OPUTX

Puts (copies) a sequential data set, a data set member, an MVS partitioned data set, or a PDSE into a z/OS UNIX directory. You can specify text or binary data, select code page conversion for single-byte data, specify a copy to lowercase file names, and append a suffix to the member names when they become file names.

OGET

Gets a z/OS UNIX file and copies it into an MVS sequential data set or partitioned data set member. You can specify text or binary data, and select code page conversion for single-byte data.

OGETX

Gets a z/OS UNIX file or directory and copies it into an MVS partitioned data set, PDSE, or sequential data set. You can specify text or binary data, select code page conversion for single-byte data, allow a copy from lowercase file names, and delete one or all suffixes from the file names when they become PDS member names.

OCOPY

Copies data in either direction between an MVS data set and a z/OS UNIX file, using ddnames. OCOPY can also copy within MVS (one data set to another data set) or within the shell (one file to another file). OCOPY has a CONVERT operand for converting single-byte data from one code page to another.

For examples of using these commands, see:

- "Using OPUT and OCOPY to copy a PDS member, a PDSE member, or a sequential data set" on page 253
- "Using OPUTX to copy a sequential data set or members of a PDS or PDSE" on page 256
- "Copying an MVS VSAM data set to a z/OS UNIX file" on page 257
- "Using OGET and OCOPY to copy a file into a sequential data set or a PDS member" on page 258
- "Copying z/OS UNIX files into a PDS or PDSE" on page 261
- "Copying an MVS data set into another MVS data set" on page 263
- "Using TSO/E commands and JCL to copy executables" on page 264

You can also invoke BPXCOPY as a TSO/E command as described in the BPXCOPY command description in *z/OS UNIX System Services Command Reference*, but the OPUT interface is generally more appropriate.

For information about the TSO/E OPUT, OPUTX, OGET, OGETX, and OCOPY commands, see <u>z/OS UNIX</u> System Services Command Reference.

For information about the TSO/E ALLOCATE and FREE commands, see <u>z/OS TSO/E Command Reference</u>. These commands have z/OS UNIX keyword parameters. It is a good idea to use the TSO/E FREE command to free the allocated data set when you have finished copying to or from a data set.

Copying a sequential data set or PDS member into a z/OS UNIX file

You might want to copy an MVS sequential data set or a member of a partitioned data set or PDSE to a z/OS UNIX file, so that:

- The data can be used by a program running under the shell.
- If it is a C program source file developed at your workstation, you can compile, link-edit, and debug it in the shell using the c89/cc/c++ and dbx commands.

The data set can be text or binary.

Using cp to copy a sequential data or PDS member into a z/OS UNIX file

The following examples use the cp command to copy a sequential data set or PDS member into a z/OS UNIX file. You use the same syntax for the mv command.

To copy an MVS sequential dataset to a z/OS UNIX file (in the current working directory):

cp "//'posix.mylogfile'" mylogfile

To copy an MVS PDS member to a z/OS UNIX file (in the current working directory):

cp "//'posix.cpmvtest(myfile)'" myunixfile

If there is an existing z/OS UNIX file with the path name that you specify on the command, it is automatically replaced and the mode of the file is not changed. The directories specified in the path name must already exist. This command creates a new file, but it does not create a new directory.

Using OPUT and OCOPY to copy a PDS member, a PDSE member, or a sequential data set

You can use the TSO/E OPUT command or OCOPY commands to do the copy. You can enter either command:

- In TSO/E, in the shell, or in ISPF. See <u>"Entering a TSO/E command" on page 185</u> for information about entering TSO/E commands in TSO/E, the shell, and ISPF.
- In batch, using a Terminal Monitor Program (TMP) job.

Using OPUT

To specify data set names and file names, use the OPUT command. To specify ddnames, use the ALLOCATE command and the OCOPY command together. Because you can specify permissions on the ALLOCATE command first, the OCOPY command lets you set the permission bits for a newly created file.

If you are moving the data set permanently to the file system, use the TSO/E DELETE command to delete the data set after copying it.

You can use the CONVERT option for single-byte data, but not for double-byte data. See <u>"Double-byte</u> data" on page 267 for information about code page conversion for double-byte data.

Example: Using OPUT with a PDSE member



If the user ID TURBO wants to copy a member of a PDSE into a file, TURBO enters the following TSO/E OPUT command:

OPUT WORKLOAD.TOTALS(OCT17) '/u/turbo/wkld/totals/oct17' TEXT CONVERT(YES)

This command:

- Copies the MVS partitioned data set member OCT17 from the data set TURBO.WORKLOAD.TOTALS to a text file with the path name /u/turbo/wkld/totals/oct17.
- Converts the data using the default conversion table (from MVS code page IBM-037 to code page IBM-1047), because YES was specified. To use a different conversion table, specify its name—for

example, BPXFX311—for conversion to and from the ASCII conversion table. If you do not want conversion, omit the CONVERT operand or specify CONVERT(NO).

For more information, see "Copying data: Code page conversion" on page 266.

• Sets a default mode (read-write-execute permission) if oct17 is a new file. For a new text (non-U-format data set) file, the default is octal 600:

```
owner=rw-
group=---
other=---
```

The default mode for a binary load module (U-format data set) is octal 700:

```
owner=rwx
group=---
other=---
```

After the file is created, you can change the permissions with the chmod command.

If there is an existing z/OS UNIX file with the path name that you specify on the command, it is automatically replaced and the mode of the file is not changed.

The directories specified in the path name must already exist. This command creates a new file, but it does not create a new directory.

Example: Using OPUT with a sequential data set

If the user ID TURBO wants to copy a sequential data set into a file, TURBO enters the following TSO/E OPUT command:

```
OPUT WORKLOAD.PROJA.NOV '/u/turbo/wkld/proja/nov' TEXT CONVERT(YES)
```

This command:

- Copies the MVS sequential data set TURBO.WORKLOAD.PROJA.NOV to a text file with the path name /u/turbo/wkld/proja/nov.
- Converts the data from the MVS code page IBM-037 to code page IBM-1047, using the default conversion table because YES was specified.
- Because **nov** is a *new* text file, this command sets a default mode (read-write-execute permission) of octal 600, representing:
 - owner=rw-
 - group=---
 - other=---

Using OCOPY

To copy a data set into a file and use data definition names (ddnames) instead of a data set name and path name, use the OCOPY command.

1. If the data set and file are not yet allocated, allocate them and specify ddnames, using either the ALLOCATE command or the DD statement in JCL.

The ALLOCATE command has these operands for allocating a z/OS UNIX file:

PATH
PATHDISP
PATHMODE
PATHOPTS

They are explained in *z/OS TSO/E Command Reference*.

2. Enter the OCOPY command, making sure that the ddnames used match the ddnames that were specified when the data set and file were allocated.

- 3. You can use the CONVERT option for single-byte data, but not for double-byte data. See <u>"Double-byte</u> data" on page 267 for information about code page conversion for double-byte data.
- 4. If you are moving the data set or partitioned data set member permanently to the file system: After the copy is completed, delete the original using the TSO/E DELETE command.

Example: Using ALLOCATE and OCOPY



1. Using the ALLOCATE command to associate the PDSE member with the ddname specified in the DDNAME keyword, user TURBO could enter:

ALLOCATE DDNAME(MVSWORK) DSNAME('TURBO.WORKLOAD.TOTALS(OCT17)')

For an ALLOCATE that begins with your TSO/E prefix as the high-level qualifier, you can enter the data set name more simply as DSNAME (WORKLOAD.TOTALS(0CT17)) — without the user ID. (The TSO/E prefix defaults to your user ID, but it can be set with the PREFIX command.) If you do not enclose the data set name in quotes, TSO/E automatically prefixes the name with your TSO/E prefix. For JCL, you need the user ID.

2. Using the ALLOCATE command to create a new z/OS UNIX file and associate it with the ddname specified in the DDNAME keyword, TURBO could enter:

```
ALLOCATE DDNAME(OPNWORK) PATH('/u/turbo/wkld/totals/oct17')
PATHDISP(KEEP,DELETE) PATHOPTS(ORDWR,OCREAT)
PATHMODE(SIRUSR,SIWUSR)
```

In this example:

- PATHDISP(KEEP,DELETE) indicates that the file should be saved if the session ends normally, but that it should be deleted if the session ends abnormally.
- The PATHOPTS operand is required only when you are creating a new file.
 PATHOPTS(ORDWR,OCREAT) indicates that the owner has read/write access and this is a new file being created.
- Specifying PATHMODE is required only when you are creating a new file (OCREAT).
 PATHMODE(SIRUSR,SIWUSR) indicates that the owner has read and write permission. If you do not specify a PATHMODE, the default permissions set when the file is allocated are:

```
owner=---
group=---
other=---
```

3. After the data set and file have been allocated, TURBO would enter the OCOPY command, using the ddnames, to copy the MVS partitioned data set member to a z/OS UNIX file using the default conversion table:

```
OCOPY INDD(MVSWORK) OUTDD(OPNWORK) TEXT CONVERT(YES) PATHOPTS(USE)
```

PATHOPTS(USE) indicates that TURBO wants to use the PATHOPTS specified on the ALLOCATE command.

Example: Using JCL and OCOPY

Alternatively, TURBO could specify the ddnames in the DD statements and perform the OCOPY in the JCL for a batch job. A DD statement allocates a data set or file and sets up a ddname. In the following

example, the //INMVS statement refers to the input data set, and the //OUTHFS statement refers to the output file:

```
//TEST JOB MSGLEVEL=(1,1)
//COPYSTEP EXEC PGM=IKJEFT01
//INMVS DD DSN=TURBO.WORKLOAD.TOTALS(OCT17),DISP=SHR
//OUTHFS DD PATH='/u/turbo/wkld/totals/oct17',
// PATHDISP=(KEEP,DELETE),
// PATHOPTS=(OWRONLY,OCREAT,OEXCL),PATHMODE=(SIRUSR,SIWUSR)
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
OCOPY INDD(INMVS) OUTDD(OUTHFS) TEXT CONVERT(YES) PATHOPTS(USE)
/*
```

In this example:

- IKJEFT01 is the name of the Terminal Monitor Program (TMP), which needs to be started to process the TSO/E OCOPY command.
- For CONVERT(YES), the default is TO1047 when you are copying from an MVS data set to a file.
- PATHOPTS(USE) indicates that TURBO wants to use the PATHOPTS specified on the ALLOCATE command.

For more information about:

- The OPUT and OCOPY commands, see *z/OS UNIX System Services Command Reference*.
- The ALLOCATE command, see *z/OS TSO/E Command Reference*.
- The FREE command, see z/OS TSO/E Command Reference.
- The JCL, see <u>z/OS MVS JCL Reference</u>.

Copying a PDS or PDSE to a z/OS UNIX directory

This topic tells you how to copy a PDS or PDSE into a z/OS UNIX directory.

Using cp to copy a PDS to a z/OS UNIX directory

The following example uses the cp command to copy a sequential data set or PDS member into a z/OS UNIX file. You use the same syntax for the mv command.

To copy all members from the fully-qualified PDS 'turbo.gammalib' to the existing z/OS UNIX directory dir1, enter the following:

cp "//'turbo.gammalib'" dir1

Note that dir1 is in the current working directory.

Using OPUTX to copy a sequential data set or members of a PDS or PDSE

The OPUTX command is actually an exec that calls OPUT. You can use the OPUTX command to copy either of these:

- Members of an MVS partitioned data set or PDSE to a z/OS UNIX directory
- A sequential data set or a single member of a partitioned data set to a file

For the copy, you can specify whether this is text or binary data, or select code page conversion. When copying a partitioned data set or PDSE, you can specify a copy to lowercase file names and append a suffix to the member names when they become file names.

You can use the CONVERT option for single-byte data, but not for double-byte data. See <u>"Double-byte</u> data" on page 267 for information on code page conversion for double-byte data.

The single quotation marks around the directory name and file name are optional. Avoid using OPUTX with path names that contain quotes and spaces. For details on the OPUTX command, see <u>z/OS UNIX</u> System Services Command Reference.



User TURBO wants to copy the members from the data set TURBO.PCALCLIB into the directory /u/turbo/pcalclib. He issues the command:

OPUTX PCALCLIB /u/turbo/pcalclib LC CONVERT(YES) SUFFIX(c)

This command:

- Copies the partitioned data set to a directory. Because the data set name is not enclosed in single quotation marks, the system automatically uses the data set whose high-level qualifier is the user's user ID.
- Converts data set member names to lowercase file names.
- Converts the file to code page IBM-1047.
- Appends the suffix . c to each file name.

The members of the partitioned data set become files in the directory:

Member name	File name
TURBO.PCALCLIB(PGM1)	/u/turbo/pcalclib/pgm1.c
TURBO.PCALCLIB(PGM2)	/u/turbo/pcalclib/pgm2.c
TURBO.PCALCLIB(LIST)	/u/turbo/pcalclib/list.c

Copying an MVS VSAM data set to a z/OS UNIX file

To copy a VSAM data set:

- 1. Use the access method services (AMS) utility to move the VSAM data set to a sequential data set.
- 2. Copy the MVS sequential data set to a z/OS UNIX file. See <u>"Copying a sequential data set or PDS</u> member into a z/OS UNIX file" on page 252 for instructions.

To move the VSAM data set to a z/OS UNIX file permanently, delete the data set from MVS with the TSO/E DELETE command.

Copying a z/OS UNIX file into a sequential data set or PDS member

You might want to copy a z/OS UNIX file to a sequential data set or to a member of a partitioned data set or PDSE. After it is moved, the file:

• Can be data for an existing MVS application program.

• Can be sent to another system, including a workstation.

You can copy text files or binary files. See <u>"Copying an executable module from the file system" on page</u> 265 for more information about copying an executable.

Using cp to copy a z/OS UNIX file into a sequential data set or PDS member

The following examples use the cp command to copy a z/OS UNIX file into a sequential data set or PDS member. You use the same syntax for the mv command.

To copy the z/OS UNIX file myunixfile (from the current working directory) to the MVS PDS member myfile within the PDS called 'posix.cpmvtest':

```
cp myunixfile "//'posix.cpmvtest(myfile)'"
```

To copy the z/OS UNIX file file1 to a new, fully-qualified sequential data set 'turbo.gammalib' to be created with specific attributes:

```
cp -P "RECFM=U,space=(500,100)" file1 "//'turbo.gammalib'"
```

To copy the z/OS UNIX file f1 to a fully-qualified sequential data set 'turbo.gammalib' and treat it as binary:

cp -F bin f1 "//'turbo.gammalib'"

Using OGET and OCOPY to copy a file into a sequential data set or a PDS member

You can use the TSO/E OGET command or the TSO/E OCOPY command to copy a z/OS UNIX file into a sequential data set or PDS member. You can enter either of these commands:

- In TSO/E, in the shell, or in ISPF. See <u>"Entering a TSO/E command" on page 185</u> for information about entering TSO/E commands in TSO/E, the shell, and ISPF.
- In batch, using a Terminal Monitor Program (TMP) job.

To work with data set names and file names, use the OGET command. To work with ddnames, use the OCOPY command.

OGET

You can use the CONVERT option for single-byte data, but not for double-byte data. See <u>"Double-byte</u> data" on page 267 for information about code page conversion for double-byte data.

Example: Using OGET with a PDSE member



If a person with the user ID TURBO enters the following command:

OGET '/u/turbo/wkld/totals/oct17' WORKLOAD.TOTALS(OCT17) CONVERT(YES)

the system:

- Copies the text file /u/turbo/wkld/totals/oct17 into the member OCT17 of the PDSE TURBO.WORKLOAD.TOTALS. (The default file type is a text file.)
- Converts the data from code page IBM-1047 to the MVS code page IBM-037, using the default conversion table. You can specify a table name if you do not want to use the default table. If you do not want conversion, omit the CONVERT operand.

For more information, see "Copying data: Code page conversion" on page 266.

If a member by this name already exists in the data set, it is replaced. If the member does not exist, a new member is created. However, if a partitioned data set or PDSE does not exist, it is not allocated.

If you are moving the z/OS UNIX file permanently to an MVS data set, remove it from the file system with the rm shell command.

Example: Using OGET with a sequential data set

If a person with the user ID TURBO enters the following command:

```
OGET '/u/turbo/wkld/proja/nov' WORKLOAD.PROJA.NOV CONVERT(YES)
```

the system:

- Copies the text file **/u/turbo/wkld/proja/nov** into the sequential data set TURBO.WORKLOAD.PROJA.NOV. (The default file type is a text file.)
- Converts the data from code page IBM-1047 to the MVS code page IBM-037, using the default conversion table. You can specify a table name if you do not want to use the default table. If you do not want conversion, omit the CONVERT operand.

For more information, see "Copying data: Code page conversion" on page 266.

If a data set with this name already exists, it is replaced. If the sequential data set does not exist, it is automatically allocated. For details on the format and size of the data set that is allocated, see the OGET command description in *z*/OS UNIX System Services Command Reference.

If you are moving the z/OS UNIX file permanently to an MVS data set, remove it from the file system with the rm shell command.

OCOPY

To copy a z/OS UNIX file into an MVS data set using data definition names (ddname) instead of a data set name or path name, use the OCOPY command.

1. If the file and data set are not yet allocated, allocate them and specify ddnames, using either the TSO/E ALLOCATE command or the DD statement for JCL.

The ALLOCATE command has these keywords for allocating a z/OS UNIX file:

PATH PATHDISP PATHMODE PATHOPTS

They are explained in *z/OS TSO/E Command Reference*.

- 2. Enter the OCOPY command, making sure that the ddnames used match the ddnames that were specified when the data set and file were allocated.
- 3. You can use the CONVERT option for single-byte data, but not for double-byte data. See <u>"Double-byte</u> data" on page 267 for information on code page conversion for double-byte data.
- 4. After the copy is completed, you can delete the file using the rm shell command.

Example: Using ALLOCATE and OCOPY



1. Using the ALLOCATE command to associate an existing z/OS UNIX file with the ddname specified in the DDNAME keyword, user TURBO could enter:

```
ALLOCATE DDNAME(OPNWORK) PATH('/u/turbo/wkld/totals/oct17')
PATHOPTS(ORDWR,OAPPEND) PATHDISP(KEEP,KEEP)
```

In this example:

- The file already exists, and PATHOPTS(ORDWR,OAPPEND) indicates that the file owner has read/ write access to the file and the owner's data should be written at the end of the file.
- PATHDISP(KEEP,KEEP) indicates that the file will be saved in case of normal or abnormal termination.
- 2. Using the ALLOCATE command to associate the output data set with the ddname specified in the DDNAME keyword, user TURBO could enter:

ALLOCATE DDNAME(MVSWORK) DSNAME('TURBO.WORKLOAD.TOTALS(OCT17)') OLD

where the DDNAME keyword specifies the ddname. OLD indicates that this is an existing data set and others cannot access the data set while the system is writing to it.

Tip: For an ALLOCATE, you can enter the data set name more simply as DSNAME (WORKLOAD.TOTALS(OCT17)) without the user ID. (TSO/E automatically prefixes the data set name with your user ID if you do not enclose the name in quotes.) For JCL, you need the user ID.

3. TURBO then enters the OCOPY command, using ddnames, to copy the z/OS UNIX file to an MVS data set:

```
OCOPY INDD(OPNWORK) OUTDD(MVSWORK) TEXT CONVERT(YES) PATHOPTS(USE)
```

PATHOPTS(USE) indicates that TURBO wants to use the PATHOPTS specified on the ALLOCATE command.

Example: Using JCL and OCOPY

Alternatively, TURBO could specify the ddnames in the //IN DD and //OUT DD statements in the JCL for a batch job. A DD statement allocates a data set or file and sets up a ddname. For example:

```
//TEST JOB MSGLEVEL=(1,1)
//COPYSTEP EXEC PGM=IKJEFT01
//INHFS DD PATH='/u/turbo/wkld/totals/oct17',PATHOPTS=(ORDONLY)
//OUTMVS DD DSN=TURBO.WORKLOAD.TOTALS(OCT17),DISP=OLD
//SYSTSIN DD SYSOUT=*
//SYSTSIN DD *
OCOPY INDD(INHFS) OUTDD(OUTMVS) TEXT CONVERT(YES) PATHOPTS(USE)
/*
```

In this example:

- IKJEFT01 is the name of the Terminal Monitor Program (TMP), which needs to be started to process the TSO/E OCOPY command.
- PATHOPTS(USE) indicates that TURBO wants to use the PATHOPTS specified on the ALLOCATE command.

For further information about:

- The OGET and OCOPY commands, see *z*/OS UNIX System Services Command Reference.
- The ALLOCATE command, see *z/OS TSO/E Command Reference*.
- The JCL, see *z/OS MVS JCL Reference*.

Copying z/OS UNIX files into a PDS or PDSE

You might want to copy z/OS UNIX files int a partitioned data set or a PDSE. After they are moved, the files:

- Can be data for an existing MVS application program.
- Can be sent to another system, including a workstation.

Using cp to copy z/OS UNIX files into a PDS or PDSE

The following example uses the cp command to copy z/OS UNIX files (in the current working directory) into a PDS or a PDSE. You use the same syntax for the mv command. This example assumes the z/OS UNIX directory does not contain subdirectories.

To drop .c suffixes before copying the files file1.c, file2.c, and file3.c in the directory dir1 into the existing PDS 'turbo.gammalib', enter the following:

cp -S d=.c file1.c file2.c file3.c "//'turbo.gammalib'"

Using OGETX to copy files into a PDS or PDSE

The OGETX command is actually an exec that calls OGET. You can use the OGETX command to copy either of these:

- Files from a z/OS UNIX directory to an MVS partitioned data set or PDSE
- An individual file to a sequential data set or member of a partitioned data set

For the copy, you can specify text or binary data and select code page conversion. When copying a directory, you can specify a copy from lowercase file names and delete one or all suffixes from the file names when they become PDS member names. For a file to be copied, its name must conform to partitioned data set member name conventions after any suffix and LC processing is done. Member names can be 1–8-character uppercase alphanumeric or national characters (A–Z, 0–9, \$, #, @). They cannot start with a numeric.

If you specify a particular suffix, only files with that suffix are copied—with the suffix deleted. If you use the SUFFIX operand without specifying a particular suffix, any file names with suffixes have the suffix deleted, and all files are copied. (After the suffix is deleted, if more than one file has the same name, each subsequent file that is copied overlays a file with the same name that was copied previously.)

The single quotation marks around the directory name and file name are optional. Avoid using OGETX with pathnames that contain quotes and spaces.

You can use the CONVERT option for single-byte data, but not for double-byte data. See <u>"Double-byte</u> data" on page 267 for information on code page conversion for double-byte data.

If the OGETX command creates a new data set, it has the same format and size as a data set created by the OGET command. For details on the OGETX command, see <u>z/OS UNIX System Services Command</u> *Reference*.

Example: Using OGETX with a PDSE



User TURBO wants to copy the directory /u/turbo/gammalib into the partitioned data set TURBO.GAMMALIB. He issues the command:

OGETX /u/turbo/gammalib GAMMALIB LC SUFFIX

This command:

- Copies into the partitioned data set all the files in the directory that meet MVS member name requirements. Because the data set name is not enclosed in single quotation marks, the system automatically supplies the user's user ID (TURBO) as a high-level qualifier.
- Copies from files with lowercase, uppercase, or mixed-case names.
- Removes any suffixes to the file names. (After the suffix is deleted, if more than one file has the same name, each subsequent file that is copied overlays a file with the same name that was copied previously.)

The files in the directory become partitioned data set members:

File name	Member name
/u/turbo/gammalib/pgm1.c	TURBO.GAMMALIB(PGM1)
/u/turbo/gammalib/pgm2.c	TURBO.GAMMALIB(PGM2)
/u/turbo/gammalib/list.prg	TURBO.GAMMALIB(LIST)

Copying files within the z/OS UNIX file system

You can use the shell commands cp or pax or the TSO/E command OCOPY to copy files within the z/OS UNIX file system.

Using the shell: Use the cp shell command to copy:

- One file to another file in the working directory, or to a new file in another directory.
- A file, a set of files, or a set of directories to another location in your file system.

To copy one file to another file in the working directory, enter:

cp file1 file2

This command copies the contents of *file1* into *file2*.

To copy a list of files into another directory, enter:

cp file1 file2 file3 dir1

This command copies the files *file1*, *file2*, *file3* into the directory *dir1*.

For further information about the cp command, see z/OS UNIX System Services Command Reference.

You can use the pax shell command in copy mode to copy a set of directories and files to another place in your file system.

To use pax in copy mode, specify the -r and -w (or -rw) options, as follows:

pax -rw pathname directory

pax reads the specified path name and copies it to the target directory. The target directory must already exist and you must have write access to it. If a path name is a directory, pax copies all the files and subdirectories in that directory, as well as the directory itself, to the target directory.

Using pax in copy mode with additional options such as -C and -M can be useful for migrating data from one file system type to another (for instance, from HFS to zFS). For further information about the pax command, see the pax command description in *z/OS UNIX System Services Command Reference*.

Using TSO/E: You can use the TSO/E OCOPY command to copy a z/OS UNIX file to another z/OS UNIX file and, in the process, convert the data from one code page to another.

Example: To copy a z/OS UNIX file to another z/OS UNIX file in a different directory, converting the data:

ALLOCATE DDNAME(KPAYR) PATH('/u/kinn/bin/payroll') ALLOCATE DDNAME(MPAYR) PATH('/u/mills/bin/payroll') OCOPY INDD(KPAYR) OUTDD(mpayr) TEXT CONVERT((BPXFX311)) T01047

The combination of CONVERT((BPXFX311)) and TO1047 indicates that you want to use the ASCII conversion table to convert from ASCII to code page IBM-1047. TO1047 or FROM1047 is required if CONVERT is specified.

With the CONVERT parameter, you can specify a data set name, a member name, or both. In this example, the use of (()) with no data set name indicates that you are specifying a member that is a module in the standard search order for MVS.

If the files that are being allocated are new files, the PATHOPTS and PATHMODE operands are required.

Copying an MVS data set into another MVS data set

You can use the TSO/E OCOPY command to copy an MVS data set into another data set. It has a CONVERT option that lets you convert between these code pages:

- IBM-037 and IBM-1047
- IBM-037 and ISO8859-1
- Code pages in a user-defined conversion table

With the TSO/E OCOPY command, you can copy:

- A sequential data set to a sequential data set
- A sequential data set to a partitioned data set or PDSE member
- A partitioned data set or PDSE member to a partitioned data set or PDSE member
- A partitioned data set or PDSE member to a sequential data set

You can enter the command:

- In TSO/E, in the shell, or in ISPF. See <u>"Entering a TSO/E command" on page 185</u> for information about entering TSO/E commands in TSO/E, the shell, and ISPF.
- In batch, using a Terminal Monitor Program (TMP) job.

The OCOPY command uses ddnames instead of data set names:

```
OCOPY INDD(ddname1) OUTDD(ddname2)
    {TEXT | BINARY}
```

```
{CONVERT(convert_table_name | YES | NO)}
{T01047 | FROM1047}
```

You do not need the PATHOPTS operand when copying from one data set to another.

There are two ways to specify ddnames, using either the ALLOCATE command or JCL for a batch job.

Example: Using ALLOCATE and OCOPY

Using the ALLOCATE command to associate each data set with a ddname, user TURBO could enter:

ALLOCATE DDNAME(TMP1) DSNAME(TEMP1) SHR ALLOCATE DDNAME(TMP10C) DSNAME(TEMP10C) OLD

where the DDNAME keyword specifies the ddname. SHR indicates that this is an existing data set and others can access it while the system is reading from it. OLD indicates that this is an existing data set and others cannot access the data set while the system is writing to it.

TURBO could then enter the OCOPY command, using the ddnames from the ALLOCATE command, to convert the data in TEMP1 from the MVS country-extended code page to code page IBM-1047, and copy it to the data set TEMP10C:

OCOPY INDD(TMP1) OUTDD(TMP1OC) TEXT CONVERT(YES) T01047

If CONVERT is specified, you must also specify TO1047 or FROM1047.

Example: Using JCL and OCOPY

Alternatively, TURBO could specify the ddnames in the //IN DD and //OUT DD statements in the JCL for a batch job. A DD statement allocates a data set or file and sets up a ddname. For example:

```
//TEST JOB MSGLEVEL=(1,1)
//COPYSTEP EXEC PGM=IKJEFT01
//IN DD DSN=TURB0.TEMP1,DISP=SHR
//OUT DD DSN=TURB0.TEMP10C,DISP=OLD
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
OCOPY INDD(IN) OUTDD(OUT) TEXT CONVERT(YES) T01047
/*
```

In this example,

- IKJEFT01 is the name of the Terminal Monitor Program (TMP), which needs to be started to process the TSO/E OCOPY command.
- TO1047 is required, because you are copying from one data set to another data set.

Copying executable modules between MVS data sets and the z/OS UNIX file system

Using cp to copy executables between MVS and z/OS UNIX

The following example uses the cp command to copy an MVS executable module to a z/OS UNIX executable file in the current working directory. You use the same syntax for the mv command.

cp -X "//'posix.my.loadlib(myexec)'" myexec

The following example uses the cp command to copy a z/OS UNIX executable file in the current working directory to an MVS executable module. You use the same syntax for the mv command.

cp -X myexec "//'posix.my.loadlib(myexec)'"

Using TSO/E commands and JCL to copy executables

Copying an executable module from a PDSE

If the load module is in a PDSE, you can copy it to the file system using one of these commands:

- The OPUT or OPUTX command. For a new text (non-U-format data set) file, the default permission is octal 600; you can use the chmod command or the MODE keyword on the OPUTX command to make it executable. If you replace an existing file, the existing permissions are unchanged.
- The OCOPY command. Specify PATHMODE(SIRWXU) to make the file executable for the owner.

Copying an executable module from a PDS

If the load module is in a partitioned data set (PDS), you can do one of these:

• Use OPUTX. If the source data set is a PDS with an undefined record format, OPUTX treats the members as load modules. In order for the program to be able to run from the file hierarchy, the entry point must be at the beginning of the load module.

In order for OPUTX to treat the file as a load module, do not specify BINARY or TEXT. Once the module is in the file system, use chmod to make it executable. If you replace an existing file, the permissions are unchanged.

• Use JCL that invokes the binder before you copy the module into the file system. See the following example for sample JCL.

Example: Using JCL to copy from a PDS to the file system

To copy a load module out of a partitioned data set and into the file system, you have to use the binder to flatten the load module. Here is an example of JCL our friend TURBO wrote for copying a z/OS XL C/C++ load module into the file system:

```
//TURBO
             JOB (XX, YY, ZZ), MSGCLASS=H, CLASS=A,
             MSGLÈVEL=(1,1)
11
1/*
//LKED
             EXEC PGM=IEWBLINK, REGION=500K,
             PARM='LIST, REUS, RENT, NCAL, LET, MAP, CASE=MIXED'
//SYSPRINT DD SYSOUT=*
//INLIB DD DSN=TURBO.POSIX.LOADLIB,DISP=SHR
//*
//SYSLMOD DD PATH='/u/turbo/llib/payrll'
//SYSLIN
          DD *
        INCLUDE INLIB(PAYRLL)
        ENTRY CEESTART
/*
```

This job relinks, or rebinds, the load module PAYRLL from TURBO.POSIX.LOADLIB(PAYRLL), and puts the output into the file system as /u/turbo/llib/payrll. Be sure you specify the correct entry point—in this case, CEESTART—for a z/OS XL C/C++ program. If you do not specify the entry point, the entry point is assumed to be at the beginning of the load module.

If the file does not exist you must specify the PATHOPTS and PATHMODE parameters on the DD statement in order to create the file with the appropriate permissions. If the file already exists but you do not have the appropriate file permissions, either the permissions or your access privileges will have to be changed.

Copying an executable module from the file system

There are two methods for copying an executable module from the file system into a data set. These methods are not exactly copy operations; instead, they bind the executables over to the data set. As a result, certain attributes are not preserved, but rather, re-established:

• Use OGETX without the TEXT or BINARY option. If you are copying into a target data set that is a PDS or PDSE with an undefined record format, OGETX treats the files as executables. Because the entry point is re-established, this option only works if the original entry point is at the beginning of the executable.

Along with the entry point, other attributes are re-established, such as the authorization (AC) value, which is reset to AC=1 (the AUTH option allows it to be set to AC=1).

Be aware that most executables will not copy successfully into a PDS. When you attempt to execute this kind of operation, you'll receive the following type of error:

IEW2606S 4B39 MODULE INCORPORATES PROGRAM MANAGEMENT 3 FEATURES AND CANNOT BE SAVED IN LOAD MODULE FORMAT.

Instead of copying into a PDS, executables created with OS/390 V2R4 or later can be copied into a PDSE.

Another exception is DLL-enabled executables created with OS/390 V2R4 or later. These executables will not copy successfully into both PDSs and PDSEs. This occurs because the information that is provided by the IMPORT control statements is not preserved, and must be specified again during the rebind.

• Use JCL that invokes the binder before you copy the executable into the data set. See the following example for sample JCL.

Example: Using JCL to copy from the file system to a PDS

To copy an executable out of the file system and into a data set, you need to use the binder to reprocess the executable. Here is an example of JCL our friend TURBO wrote for copying a z/OS XL C/C++ load module into a data set:

```
//TURBO
            JOB (XX,YY,ZZ),MSGCLASS=H,CLASS=A,
            MSGLEVEL=(1,1)
//
//*
            EXEC PGM=IEWBLINK, REGION=500K,
//LKED
            PARM='LIST, REUS, RENT, NCAL, LET, MAP'
//SYSPRINT DD SYSOUT=*
//SYSLMOD DD DSN=TURBO.POSIX.LOADLIB,DISP=SHR
//*
//INLIB
           DD PATH='/u/turbo/llib/payrll',
               PATHOPTS=(ORDONLY)
//
//*
//SYSLIN
           DD
               *
        INCLUDE INLIB
        ENTRY CEESTART
        NAME PAYRLL(R)
/*
```

This job relinks, or rebinds, the XL C/C++ executable /u/turbo/llib/payrll, and puts the output into TURBO.POSIX.LOADLIB(PAYRLL). Be sure you specify the correct entry point—in this case, CEESTART—for a XL C/C++ program. If you do not specify the entry point, the entry point is assumed to be at the beginning of the executable. Also, if required, you must specify AC=1.

If this is a DLL created using V2R4 or later and without the use of the prelinker, any definition side-decks of IMPORT control statements will need to be re-specified as input to the binder. In general, any control statements and options that were used when the original /u/turbo/llib/payrll was created will need to be specified again.

Copying data: Code page conversion

The method you use to convert data from one code page to another depends on whether it is single-byte or double-byte data.

Single-byte data

If you are copying single-byte data into or out of the z/OS UNIX file system, you can use one of these:

- Working in MVS, you can use the z/OS XL C/C++ iconv utility to convert MVS data from one code page to another. For information about the z/OS XL C/C++ iconv utility, see <u>z/OS XL C/C++ Programming</u> <u>Guide</u>.
- Working in the shell, you can use the iconv shell command to convert z/OS UNIX data from one code page to another. For information about the iconv shell command, see the iconv command description in z/OS UNIX System Services Command Reference.
• The CONVERT operand on the OCOPY, OGET, OGETX, OPUT, and OPUTX commands provides these code page conversion choices for the data as you are copying:

CONVERT((BPXFX111))

Specifies a conversion table to convert between code pages IBM-037 and IBM-1047.

CONVERT((BPXFX311))

Specifies an ASCII-EBCDIC conversion table to convert between code pages ISO8859-1 and IBM-1047.

CONVERT(YES)

Specifies the default conversion table BPXFX000, which is an alias that points to BPXFX111, to convert the data.

CONVERT(user-defined table)

Specifies the name of a user-defined conversion table.

In this list, the use of (()) with no data set name indicates that you are specifying a member that is a module in the standard search order for MVS.

Double-byte data

If you are moving double-byte data into or out of the z/OS UNIX file system, you can convert the data to or from the shell-supported DBCS code page IBM-939 using one of two utilities:

- Working in MVS, you can use the z/OS XL C/C++ iconv utility. For information about the z/OS XL C/C++ iconv utility, see z/OS XL C/C++ Programming Guide.
- Working in the shell, you can use the iconv shell utility. For information about the iconv shell utility, see the iconv command description in *z/OS UNIX System Services Command Reference*.

Example: Using the iconv shell utility with MBCS data

In this example, the PDSE member MBCSDATA is moved into the file system and then converted to code page IBM-939 from code page IBM-932 (a multibyte ASCII code page):

1. Run the OPUT command from the shell, using the double quotation marks to prevent the shell from processing it:

```
tso oput "'usr3.data(mbcsdata)' '/tmp/usr3/mbcsdata' bin"
```

2. Change to the directory that the file **mbcsdata** is in:

cd /tmp/usr3

3. Use iconv to convert the data and put it into the output file **dbcsdata**:

iconv -f IBM-932 -t IBM-939 mbcsdata > dbcsdata

Chapter 22. Transferring files between systems

You can create applications and files at your workstation and then move the resulting files to the z/OS UNIX file system for further application development, such as compiling and debugging, or to share the files. There may also be times when you want to send z/OS UNIX files to your workstation. This information discusses several methods for moving files directly between your workstation and z/OS UNIX. Note that most of the examples show z/OS UNIX files being transferred to or from the workstation.

File transfer directly to or from z/OS UNIX

To move a file or file system between your workstation and z/OS UNIX, you can use one of the following methods.

Transferring files using File Transfer Protocol (FTP)

If both the workstation and z/OS UNIX have TCP/IP installed, you can use the File Transfer Protocol (FTP) facility of TCP/IP.

With the z/OS Communications Server installed on a remote z/OS system, you can ftp files into or from that system's file system.

An FTP client is not available for the shell and utilities.

Transferring files using the Network File System feature

Using the Network File System feature, you can edit or browse a z/OS UNIX file directly from your workstation. For example, if you want to copy a file to a workstation file, you do not need to move it to an MVS data set first. Here is an example showing the steps involved:

- 1. Log on to the host using mvslogin.
- 2. Mount the directory /u/usr1/a/b at the workstation with the command:

mount mvshost:"/u/usr1/a/b" /x/y

3. Copy the file /u/usr1/a/b/c to the workstation file /mycopy/c with the command:

cp /x/y/c /mycopy/c

Using the Network File System feature from your workstation, you can copy a workstation file to z/OS UNIXfile without having to move it to an MVS data set first. This example assumes that you have run your mvslogin and mounted the directory /u/usr1/pgma/b at the workstation under the path name / mypgma/b. You copy the workstation file /proj2/modc to the file /u/usr1/pgma/b/modc with the command:

cp /proj2/modc /mypgma/b/modc

Suppose you have an executable that you compiled and linked on a workstation, and you want to store it in an MVS data set but run it from the workstation. You copy the executable to the mounted z/OS UNIX file in binary format. Later, when you want to run the program from the workstation, you use NFS to mount the directory in binary format, and then run the program from the mounted z/OS UNIX file system.

For more information about working with NFS files on your workstation, see <u>z/OS Network File System</u> Guide and Reference.

Transferring files using the SEND and RECEIVE programs

You can transfer files using the SEND and RECEIVE programs that are available with PC 3270 emulation programs and with OS/2 Extended Edition Version 1.2 or later.

Before you use the SEND and RECEIVE programs, you must be working in TSO/E. If you are using the OMVS interface to work in the shell, use the TSO function key to switch to TSO/E command mode before using the programs.

Transferring files by using the File Transfer, Access, and Management Function

You can also transfer files between your workstation and the z/OS UNIX file system by using the File Transfer, Access, and Management (FTAM) function of OSI/File Services.

File transfer using MVS data sets

Transferring files between systems can also take place without the Network File System feature.

Transferring files into the z/OS UNIX file system

If the z/OS Communications Server is installed on a remote system, you can ftp files directly into that file system.

Tip: If you are ftp-ing to a remote z/OS UNIX file system, be aware that the z/OS UNIX server often listens to a port other than the well-known port. Make sure you know the address and port to use.

If you are not using the Network File System feature and the z/OS Communications Server is not installed, perform these steps:

- 1. Transfer the data to the host, using your preferred method (for example, FTP).
- 2. While logged on to TSO/E, copy the data from an MVS data set into the file system, using the TSO/E OPUT command.

Single-byte data: If you need to convert to a shell-supported code page, use the CONVERT option on the OPUT command. See "Using OPUT" on page 253.

Double-byte data or multibyte ASCII-based data: If you need to convert to a shell-supported code page, use the z/OS C/C++iconv utility (while working in MVS) or the iconv shell utility (while working in the shell). For more information, see "Copying data: Code page conversion" on page 266.

3. If you want, you can delete the MVS data set after the copy with the TSO/E DELETE command.

Transferring files to the workstation

If you are working without the Network File System feature, perform these steps while logged on to TSO/E:

1. Copy the file to an MVS data set (sequential or partitioned) using the TSO/E OGET command. See "OGET" on page 258.

Single-byte data: If you need to convert to a different code page, you can use the CONVERT option on the OGET command.

Double-byte data: If you need to convert the data, you can use the iconv shell utility while working in the shell. For more information, see "Copying data: Code page conversion" on page 266.

- 2. If you want, you can delete the file after the copy with the rm shell command.
- 3. Send the data set to the workstation, using your preferred method (for example, FTP).

Transporting an archive file on tape or diskette

A directory or file system that is going to be transported on tape or diskette is put into an archive file, as discussed in <u>"Backing up and restoring files: options" on page 211</u>. This information discusses the steps involved in

- Installing an archive file from tape or diskette into a z/OS UNIX file system
- Putting an archive file on tape or diskette to send to another site

Putting an archive file into the file system

You may receive an archive file on tape or diskette. There are two main steps involved in installing the archive file into a z/OS UNIX file system:

- 1. Transferring the archive file to an MVS data set, from either a workstation or a tape drive at your MVS system.:
- 2. Copying the archive file from the data set into the file system

Step 1. Transferring the archive file to a data set

From a workstation: If you have TCP/IP on your workstation, you can use the ftp command to transfer an archive file to MVS or to the z/OS shell (if you have the z/OS Communications Server installed).

a.

Copy the archive file into a file.

b.

Enter the FTP command.

Tip: If you are ftp-ing to a remote z/OS UNIX file system, be aware that the z/OS UNIX server often listens to a port other than the well-known port. Make sure you know the address and port to use.

c.

Enter the **binary** subcommand.

d.

Enter the **put** subcommand, specifying a z/OS UNIX directory or a sequential or partitioned data set as the destination.

If you are specifying a data set, you may prefer to use one partitioned data set for all your archive files, with each archive file a member in the partitioned data set. Here is an example of the partitioned data set attributes you might want:

```
DATA SET NAME: TURBO.CMPL.ARCHIVE
                                         CURRENT ALLOCATION:
GENERAL DATA:
                          TRBLK1
   Volume serial:
                                            Allocated Cylinders: 26
   Device type:
                          3380
                                            Allocated extents:
                                                                   5
   Organization:
                          P0
   Record format:
                          VB
                          255
   Record length:
   Block size:
                          23476
                                        CURRENT UTILIZATION:
   1st extent Cylinders: 12
                                           Used Cylinders:
                                                                  0
   Secondary Cylinders:
                          0
                                            Used extents:
                                                                   0
   Creation date:
                          1994/12/18
   Expiration date:
                          ***NONE***
```

e.

Go to "Step 2. Copying the file from a data set into a file system" on page 272.

From a tape drive at your MVS system: If you have an archive file on tape and the necessary tape drive at your MVS system, you can copy the file directly from the tape into a data set.

a.

Copy the archive file from the tape into a data set. Here is some sample JCL for copying an archive file (TURBO.TARTAPE) from a tape into a data set (TURBO.TAR):

//TAPE2DS JOB ,',MSGLEVEL=(1,1) //* //STEP1 EXEC PGM=IEBGENER //SYSIN DD DUMMY

//SYSPRINT	DD	SYSOUT=*
//SYSUT1	DD	DSNAME=TURBO.TARTAPE, UNIT=TAPE, LABEL=(1, NL), DISP=OLD,
11		VOL=SER=123456,DCB=(RECFM=U,BLKSIZE=5120)
//SYSUT2	DD	DSNAME=TURBO.TAR,DISP=(NEW,CATLG),UNIT=SYSDA,
//		SPACE=(5120,(100,100),RLSE)

Note: For LABEL=, NL indicates that there is no label. Use NL when transferring a tape between an MVS and a UNIX system; use SL when transferring a tape between two MVS systems.

b.

Go to "Step 2. Copying the file from a data set into a file system" on page 272, which follows.

Step 2. Copying the file from a data set into a file system

Working in MVS: Use the pax or the tar shell command to restore the directory or file system from the archive file; all the component files are restored from the archive file. If you need to convert the source to the code page IBM-1047 used in the z/OS shell, use the pax command with the **-o** option. See <u>"Backing</u> up and restoring files: options" on page 211 for more information.

Sending an archive file to others

The following are the steps for sending an archive file that contains multiple files on tape or diskette. In the example, the pax command creates an archive file for a directory or file system. The TSO/E OGET command, with the BINARY option, then copies the archive file into a partitioned data set or a sequential data set. Step 2 is not necessary with OS/390 Release 8 and later.

Step 1. Create an archive file for multiple files

You can use either the pax or the tar shell command to create the archive file. All the component files are stored in one archive file. For more information, see <u>"Backing up and restoring files from the shell" on page 212</u>, and also see <u>z/OS UNIX System Services Command Reference</u> for a complete description of the pax and tar commands.

If you need to convert to a different code page than the one used in the shell, use the pax command with the **-o** option. See <u>"Backing up and restoring files: options" on page 211</u> for more information.

Step 2. Copy the file from the file system to a data set

Use the TSO/E OGET command with the BINARY option to copy the archive file into a sequential data set. See "OGET" on page 258 for more information.

tso "OGET '/tmp/testpgm.pax' 'POSIX.TESTPGM.PAX' BINARY"

The OGET command copies the archive file into the specified MVS data set:

- '/tmp/posix/testpgm.pax' is the absolute pathname for the archive file.
- 'POSIX.TESTPGM.PAX' is the fully qualified data set name for the data set.

• BINARY indicates that the data is binary.

The final step is to use **ftp** (or some other method) to send the file to the intended destination.

Step 3. Transfer the archive file to a tape or diskette

To a tape or diskette at the workstation: While working in MVS:

a.

For information about how to copy an archive file from the file system into a data set, see <u>"Step 2.</u> Copy the file from the file system to a data set" on page 272.

b.

Enter the **FTP** command.

c.

Enter the **binary** subcommand.

d.

Enter the **put** subcommand, specifying a pathname at your workstation as a destination.

e.

At the workstation, copy the archive file into a file.

To a tape at the host: While working in MVS:

a.

For information about how to copy an archive file from the file system into a data set, see <u>"Step 2.</u> Copy the file from the file system to a data set" on page 272.

b.

Copy the archive file from the data set to tape. Here is some sample JCL for copying a data set containing an archive file (TURBO.TAR) to a tape (TURBO.TARTAPE):

```
//DS2TAPE JOB ,',MSGLEVEL=(1,1)
//*
//STEP1 EXEC PGM=IEBGENER
//SYSIN DD DUMMY
//SYSPRINT DD SYSOUT=*
//SYSUT1 DD DSNAME=TURB0.TAR,DISP=OLD
//SYSUT2 DD DSNAME=TURB0.TARTAPE,UNIT=TAPE,LABEL=(1,NL),
// DISP=(NEW,KEEP)
```

Guideline: For LABEL=, NL indicates that there is no label. Use NL when transferring a tape between an MVS and a UNIX system; use SL when transferring a tape between two MVS systems.

Appendix A. Advanced vi topics

After you have mastered basic usage of vi, as described in <u>"Using the vi screen editor" on page 228</u>, you might want to explore some of the editor's other capabilities.

Editing options

vi has many options that change the way the editor behaves during an editing session. We will discuss a few that may be immediately useful. For a complete list of these options, see the <u>vi</u> command description in *z/OS UNIX System Services Command Reference*.

You must be in Command Mode to set options. To set an option, begin by typing a colon (:). You will see the cursor move to the bottom of the screen. Then type the word set, a space, and the name of the option you want to set—we will talk about option names in a moment. You can correct typing mistakes by backspacing. When you have typed everything correctly, press <Enter>.

One commonly used option is "ignorecase". If you type:

:set ignorecase

vi does not pay attention to the case of letters when searching. Many people prefer caseless searches over case-sensitive ones. If you want to go back to case-sensitive searches, type

:set noignorecase

Setting tab stops

By default, vi sets tab stops every 8 spaces. For example, if you begin a paragraph by typing a tab, the tab moves the cursor over 8 spaces. Many people feel 8 spaces are too many for a tab stop. You can set tab stops of 5 spaces with:

:set tabstop=5

Similar commands can set tab stops to any number of spaces.

Using abbreviations

You can define an abbreviation for commonly used words or phrases. For example, if you type:

:ab www World Wide Web

and then press <Enter>, this sets the abbreviation. As soon as you type the abbreviation in text and move the cursor to the next space after www, the abbreviation is expanded into the associated phrase. The abbreviation function is case-sensitive. An abbreviation lasts for the duration of a vi session. For information about how to set up a file with frequently used editing options, see <u>"Setting up an editing</u> options command file" on page 276.

If you want to get rid of an abbreviation that has been set, use the : una (unabbreviate) command. For example, type:

:una www

to get rid of the abbreviation.

Other editing options

For a complete list of the editing options, see the <u>vi</u> command description in <u>z/OS UNIX System Services</u> Command Reference.

Setting up an editing options command file

A command file contains a number of commands that can be executed as if they were typed in a vi session. For example, you might use vi to create a file with the contents:

set wrapmargin=8 set tabstop=5 set shiftwidth=5 ab www World Wide Web

This sets all the options you want to use and all the abbreviations you need. The file can only contain instructions that normally start with a colon (:) in vi, but you omit the colons in the command file. During a vi session, you can execute all the instructions in the command file with the instruction:

:so cmdfile

where *cmdfile* is the name of your command file. so stands for source, and it tells vi that the given file should be taken as the source of a number of commands.

You can execute the commands in a command file when you first start vi. Start vi with the command:

vi -c 'so cmdfile' filename

where *cmdfile* is the name of your command file and *filename* is the name of the file you want to edit. You might want to set up an alias for vi -c 'so cmdfile'; for example:

```
alias vic="vi -c 'so cmdfile'"
```

You can also set up a **\$HOME/.exrc** file that contains all the commands you may want to run whenever you enter vi.

Editing several files

In a typical vi session, you may want to edit several files. When you have finished editing one file, you must first save your text in that file. Once you have saved your changes, you can start editing a different file by typing:

:edit newfilename

and then press <Enter>. This will clear out the text you have been editing and set things up so you can edit the new file. If the file already exists, its current contents will be read in.

Here's a trick to remember when you want to edit a number of files. If you start vi with a command line of the form

vi file1 file2 file3 ...

you can edit several files one after the other. After you have finished editing a file and saved it, you can move among files using the following commands:

```
Command
Action
```

ACTIO

Edits the next file in the list of files.

:n!

:n

Edits the next file in the list of files and discards the changes made to the current file.

```
:n filenames
```

Specifies a new list of files to be edited.

It may be particularly useful to use wildcard characters on the vi command line, as in

vi *.c

This is expanded to a list of all the files under the current catalog that have the **.c** extension.

Combining files

Occasionally, you may want to combine a number of files into a single document. For example, you may have a table of data stored in one file and want to add the table to another file. You can read in the contents of a file after the line that holds the cursor. The r stands for Read; it reads the contents of a file to be added to the current file after the line indicated by the cursor.

The same sort of command can be used to combine the chapters of a document into a single file. For example,

```
:r chapter1
G
:r chapter2
G
:r chapter3
```

will read in chapters that are stored in separate files. Notice that we had to use G commands to go to the end of the file after each read operation, so that the next input file would be added to the end of the text.

Editing program source code

Because vi originated on a UNIX system, the editor has a number of features aimed primarily at programming in the C language. However, these same features are applicable to many other languages.

Controlling indention

The source code for a program differs from ordinary text in a number of ways. One of the most important of these is the way in which source code uses indention. Indention shows the logical structure of the program: the way in which statements are grouped into blocks.

Issue the command:

:set autoindent

(Don't forget to press<Enter>after you have typed this.) This command turns on an option supplied primarily to control indention when entering source code. Each line is automatically indented the same distance as the previous one. As a programmer, you will find this saves you quite a bit of work getting the indention right, especially when you have several levels of indention.

When you are entering code with autoindent enabled, typing <EscChar-T> gives you another level of indention, and typing <EscChar-D>takes one away. While you are in Insert Mode (not Command Mode):

- Type <EscChar-T> at the start of a line to indent it in one level.
- Type <EscChar-D> at the start of a line to indent it out one level.

The amount of indention provided by <EscChar-T> is one tab character; the space depends on the setting of tabstop.

Try using the autoindent option when you are entering source code. It simplifies the job of getting indention correct. It can even sometimes help you avoid bugs; for example, in C source code, you usually need one closing } for every level of indention you go backwards.

The << and >> commands are also helpful when indenting source code:

>>

Shifts a line right 8 spaces (that is, adds 8 spaces of indention)

<<

Shifts a line left 8 spaces (that is, removes 8 spaces of indention)

You can shift a number of lines by typing the number followed by >> or <<. For example, typing 5>> will indent five lines, including the line the cursor is on.

The default shift is 8 spaces (right or left). You can change this default with this command:

:set shiftwidth=4

Tip: It is convenient to have a shiftwidth that is the same size as the width between tab stops.

Searching for opening and closing brackets

The characters (, [, {, and < can all be called opening brackets. When the cursor is resting on one of these characters, pressing the % key moves the cursor from the opening bracket to the corresponding closing bracket character),], }, and >, keeping in mind the usual rules for nesting brackets. For example, if you move the cursor to the first (in:

```
if ( cos(a i ) > sin(b i +c i ) )
        {
            printf("cos and sin equal!");
        }
```

and press %, you will see the cursor jump to the parenthesis at the end of the line. This is the closing parenthesis that matches the opening one.

Similarly, if the cursor is on one of the closing bracket characters, pressing **%** will move the cursor backwards to the corresponding opening bracket character.

Not only does this search character help you move forward and backward through a program in long jumps, but it also lets you check the nesting of parentheses in source code. For example, if you put the cursor on the first **{** at the beginning of a C function, pressing **%** should move you to the **}** that (you think) ends the function. If it doesn't, something has gone wrong somewhere.

Making substitutions

If the name of a data object or function has to be changed in a program (for whatever reason), it becomes necessary to change every occurrence of that name. This would be a tedious process using the vi features we have discussed up to this point, because you would have to search through each source file for the name and then type in the new name wherever the old one was found. To avoid much of this work, vi offers the substitute command.

The usual form of the substitute command is

:s/pattern/replacement/

where *pattern* is any of the patterns used in searches, and *replacement* is any string.

As soon as you type the colon (:), you see the cursor move to the bottom of the screen. Then type the rest of the command and press <Enter>. The command puts the given *replacement* string in the place of the first string that matches the given *pattern*.

What happens if a line has more than one string that matches the pattern? The s command replaces only the first occurrence of a given string on a line. The position of the cursor in the line does not matter.

If you want to change every occurrence of a string on a line, type a g (for global) after the last slash.

Specifying a range of lines to change

You can also apply s to a range of lines. For example, let's examine the command:

:1,200s/^/!/

What happens? The 1, 200 in front of the s indicates that the command should be applied to the lines from 1 through 200 (everything up to the 200th line in the file). The s command itself says to replace the

beginning of the line (^) with an exclamation point. So an exclamation point would be put at the beginning of every line up to number 200. To get rid of the exclamation points, you would type:

:1,200s/^!//

which says change every ! at the beginning of a line to nothing.

Determining line numbers

In these instructions, we made use of line numbers to refer to lines. How do you know what number a line has? If you just want to know the number of one line, move the cursor to that line and type

:.=

For another approach, type:

:set number

and press <Enter>. As you can see, this displays the number of every line in the file. If you want to turn off the display of line numbers, type:

:set nonumber

A number of special symbols can be used when specifying a range of lines. The . (period) stands for the line where the cursor is currently positioned. For example, move the cursor to this line and type:

:1,.s/\$/???/

This adds ??? to the end of every line from the start of the file to the line containing the cursor.

When you issue a substitute command with a range, it is all right if some of the lines in the range do not contain the pattern you are replacing. When specifying a range of lines, \$ stands for the last line in the file. For example, the command:

:1,\$s/the/THE/g

changes every the in the file to uppercase (including words like there, where the is part of another word).

Checking as you substitute

What would you do now if you want to change the variable *i* into a k? You can't just use an instruction like

:254,267s/i/k/g

because that will change the letter *i* into *k* even in other words like int and list.

The solution to this is to add a c (for check) after the s command. For example,

:s/pattern/replacement/gc

When you do this, vi checks with you before making every substitution. Before each possible change, vi prints the line at the bottom of your screen and puts a ^ under the string that might be changed. If you want the change to happen, press the <Y> key followed by <Enter>. If you do not want the change to happen, press the <N> key followed by <Enter>.

Appendix B. Using awk

awk is a programming language that lets you work with information stored in files. With awk programs, you can:

- Display all the information in a file, or selected pieces of information
- · Perform calculations with numeric information from a file
- Prepare reports based on information from a file
- Analyze text for spelling, frequency of words or letters, and so on

You can combine these operations to perform quite complicated tasks.

awk allows most of the logical constructs of modern computing languages: if-else statements, while and for loops, function calls, and so on.

This appendix introduces some of the principles and concepts of awk. The z/OS version of awk is based on the POSIX definition of awk, and also supports the functionality of nawk, the new awk. Experienced programmers may prefer to turn directly to the awk command description in <u>z/OS UNIX System Services</u> <u>Command Reference</u>. For an excellent reference for awk, see <u>The AWK Programming Language</u> 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 <u>awk</u> 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, and punctuation characters.

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

This file is included with the z/OS UNIX shell as /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. The newline character shows where one line of text ends and another begins. By using the newline as a

record separator, each line of the file becomes a separate record. This is convenient and easy to understand; newline is used as a record separator in all of the examples.

In the hobbies file, each line is a separate record, giving a set of information about one person's hobby.

Fields

A record consists of a number of fields. A field is a single piece of information. For example, the hobby record:

Jim reading 15 100.00

contains four fields:

Jim reading 15 100.00

Fields should be provided in the same order in each record. That way awk and other programs can easily access a particular piece of information in any record.

The fields of a record are separated by one or more field separator characters. The hobbies file uses strings of blank characters (spaces) to separate fields. By default, awk uses blanks or horizontal tab characters to separate fields. You can change the default.

The shape of a program

An awk program looks like this:

```
pattern {actions}
pattern {actions}
pattern {actions}
```

Each line is a separate instruction. awk looks through the data files record by record and executes the instructions, in the given order, on each record.

Simple patterns

An 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:

Johnrole playing8100.00Johnjogging830.00

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 use the print action; however, this action does not need to be written explicitly. 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. 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 Lori	cartooning	5	75.00

awk looks through the file record by record. The first record to satisfy one of the patterns is:

Jim bridge 4 10.00

so awk prints the first field of the record (as dictated by the second instruction). The next record of interest is:

Linda bridge 12 30.00

This satisfies the first instruction's pattern, so the whole record is printed. It also satisfies the second instruction's pattern, so the first field is printed. awk continues through the file, record by record, executing the appropriate actions when a record satisfies the pattern.

Assigning values to variables

Suppose you want to find out how many people have jogging as a hobby. To do this, you have to look through the **hobbies** file, record by record, and keep a count of the number of records that have jogging in their second field. This means that you have to *remember* the count from one record to the next.

awk programs *remember* information by using *variables*. A variable is a storage place for information. Every variable has a name and a value. An awk action of the form:

name = value

assigns the specified value to the variable that has the given name. For example:

count = 0

assigns the value 0 to the variable *count*.

You can use variables in expressions. For example, the value of the expression:

count + 1

is the current value of *count*, plus 1.

String values

A *string value* is just a sequence of characters, like "abc". A string value is always enclosed in quotes. All types 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 that is 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 value is not enclosed in quotes. For example:

10 0.34 -78 +2.56 -.92

are all valid in awk. awk does not let you put commas inside numbers. For example, you must write 1000 instead of 1,000.

Note: awk lets you use exponential or scientific notation. Exponents are given as e or E, followed by an optionally signed exponent. Thus:

1E3 1.0e3 10E2 1000

are all equivalent.

When awk compares numbers (with such operators as > or <), it makes comparisons in accordance with the usual rules of arithmetic.

Using the print action for output

So far, print has been the only action discussed. As you have seen, print can display an entire record. It can also display selected fields of the record, as in:

\$2 == "bridge" { print \$1 }

This displays the first field of every record with a second field that is bridge. The output is:

Jim Linda Lori

print can display more than a single field. If you give print a list of fields separated by commas, as in:

```
$1 == "Jim" { print $2,$3,$4 }
```

print displays the given fields separated by single blanks, as in:

reading 15 100.00 bridge 4 10.00 role playing 5 70.00

The print action can display strings and numbers along with fields. For example:

```
$1 == "John" { print "$",$4 }
```

prints:

\$ 100.00 \$ 30.00

In this instruction, the print action prints a string containing a \$, followed by a blank, followed by the value of the fourth field in each selected record.

As an exercise, predict the output of the following:

(a) \$1 == "Lori" { print \$1,"spends \$", \$4,"on",\$2 }
(b) \$2 == "jogging" { print \$1,"jogs",\$3,"hours a week" }
(c) \$4 > 100.00 { print \$1, "has an expensive hobby" }

You can check your predictions by running these programs against the hobbies file.

Running awk programs

There are two ways to run awk programs: from a command line and from a program file.

The awk command line

The simplest awk command line is:

awk 'program' datafile

The awk program is enclosed in single-quote or apostrophe (') characters. The *datafile* argument gives the name of the data file. For example:

awk '\$1 == "Linda"' hobbies

executes the program:

\$1 == "Linda"

on the data file **hobbies**.

If you are using the z/OS 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 allows you to define the value of variables on the command line by using the -v option. See <u>z/OS</u> UNIX System Services Command Reference for details.

Program files

A program file is a text file that contains an awk program. You can create program files with any text editor (such as ed). For example, you might create a file named **lbprog.awk** that contains the lines:

```
$1 == "Linda"
$2 == "bridge" { print $1 }
```

To execute a program on a particular data file, use the command:

```
awk –f progfile
datafile
```

where *progfile* is the name of the file that contains the awk program, and *datafile* is the name of the data file. For example:

```
awk -f lbprog.awk hobbies
```

runs the program in **lbprog.awk** on the data in **hobbies**.

If the data file does not use the default separator characters, you must specify a –F option after the *progfile* name, as in:

awk -f prog.awk -F":" file.dat

To gain some experience using awk, you can test the examples on the **hobbies** file. Run some from the command line and some from program files.

Sources of data

If you do not specify a data file on the command line, awk begins to read data from standard input. For example, if you enter the command:

awk '{ print \$1 }'

awk prints the first word of every line you type. When you type in data from the workstation, press <Enter> at the end of each line. To stop passing data to awk, type <EscChar-D> and press <Enter>.

A command line may also specify several data files, as in:

```
awk -f progfile data1 data2 data3 ...
```

When awk has finished reading through the first data file **data1**, it goes on to **data2**, and so on.

Operators

awk recognizes these types of operators:

- Comparison operators
- · Arithmetic operators
- Compound assignments
- · Increment and decrement operators
- · Matching operators
- Multiple-condition operators

Comparison operators

The == notation is an example of a *comparison*. awk recognizes several types of comparisons:

Operator	Meaning
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Arithmetic operators

The following awk program uses simple arithmetic:

\$3 > 10 { print \$1, \$2, \$3-10 }

In the print statement:

\$3-10

has the value of the third field in the record, minus 10. This is the value that print prints. If you apply this program to the **hobbies** file, the output is:

```
Jim reading 5
Linda bridge 2
Katie jogging 4
Andrew wind surfing 10
Lori weight lifting 2
```

You could describe how the program works like this: If someone spends more than 10 hours on a hobby, the program prints the person's name, the name of the hobby, and how many *extra* hours the person spends on the hobby (that is, the number of hours more than 10).

An expression such as:

\$3-10

is called an *arithmetic expression*. It performs an arithmetic operation and comes up with a result, which is called the *value* of the expression.

awk recognizes the following arithmetic operations:

Operation	Operator	Example
Addition	A + B	2+3 is 5
Subtraction	A – B	7–3 is 4
Multiplication	A * B	2*4 is 8
Division	A / B	6/3 is 2
Negation	- A	– 9 is –9
Remainder	A % B	7%3 is 1
Exponentiation	A ^ B	3^2 is 9

The remainder operation is also known as the *modulus*, or *integer remainder* operation. The value of this expression is the integer remainder you get when you divide A by B. For example:

7 % 3

has a value of 1, because dividing 7 by 3 gives you 2 with a remainder of 1.

The value for the *exponentiation* operation:

Α ^ Β

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 the awk command description in <u>z/OS UNIX System</u> Services Command Reference.

Compound assignments

The following are the compound assignment operations of awk and their equivalents:

Compound operation	Equivalent	
A += B	A=A+B	
A -= B	A=A-B	
A *= B	A = A * B	
A /= B	A = A / B	
A %= B	A = A % B	
A ^= B	$A = A \wedge B$	

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*.

— ^sr —

The $-s_{r}$ – 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 a 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.

Ш

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 Regular expressions (regexp) in *z/OS UNIX System Services Command Reference*.

The following characters have special meanings when you use them in regular expressions:

Character Meaning

۸

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).

L

Means or. For example:

/Linda|Lori/

is a regular expression that matches either of the strings Linda or Lori.

*

Indicates zero or more repetitions of a character. For example:

/ab*c/

matches abc, abbc, abbbc, and so on. It also matches ac (zero repetitions of b). Since . matches any character except the newline, .* matches an arbitrary string of zero or more characters. For example:

\$2 ~ /^r.*g\$/ { print }

prints any record with a second field that begins with r, ends in g, and has any set of characters between (for example, reading and role playing).

+

Is similar to *****, but stands for *one* or more repetitions of a character. For example:

/ab+c/

matches abc, abbc, and so on, but does not match ac.

\{*m,n*\}

Indicates *m* to *n* repetitions of a character (where *m* and *n* are both integers). For example:

/ab\{2,4\}c/

would match abbc, abbbc, and abbbbc, and nothing else.

?

Is similar to *, but stands for zero or one repetitions of a string. For example:

/ab?c/

matches ac and abc, but not abbc, and so on.

[X]

Matches any one of the set of characters X given inside the square brackets. For example:

\$1 ~ /^[LJ]/ { print }

prints any record whose first field begins with either L or J. As a special case: [:lower:] inside the square brackets stands for any lowercase letter, [:upper:] inside the square brackets stands for any uppercase letter, [:alpha:] inside the square brackets stands for any letter, and [:digit:] inside the square brackets stands for any digit.

Thus:

```
/[[:digit:][:alpha:]]/
```

matches a digit or letter.

[^X]

Matches any one character that is not in the set X. For example:

\$1 ~ /^[^LJ] / { print }

prints any record with a first field that does not begin with L or J.

\$1 ~ /^[^[:digit:]]/ { print }

prints any record with a first field that does not begin with a digit.

(X)

Matches anything that the regular expression X does. You can use parentheses to control how other special characters behave. For example, * normally applies to the single character immediately preceding it. This means that:

/abc*d/

matches abd, abcd, abccd, and so on. However:

/a(bc)*d/

matches ad, abcd, abcbcd, abcbcbcd, and so on.

The characters with special meanings are:

^\$.*+?[]()|

These are known as metacharacters.

When a metacharacter appears in a regular expression, it usually has its special meaning. If you want to use one of these characters literally (without its special meaning), put a backslash in front of the character. For example:

/\\$1/ { print }

prints all records that contain a dollar sign \$ followed by a 1. If you simply entered:

/\$1/ { print }

awk would search for records where the end of the record was followed by a 1, which is impossible.

Because the backslash has this special meaning, $\$ is also considered a metacharacter. If you want to create a regular expression that matches a backslash, you must therefore use two backslashes $\$.

Pattern ranges

An 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 pĺaying	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 301.

Built-in variables

awk has a number of *built-in variables* that you can use in your programs. You do not have to assign values to these variables; awk automatically assigns the values for you.

Built-in numeric variables

The following list describes some of the important numeric built-in variables:

NR

Contains the number of records that have been read so far. When awk is looking at the first record, NR has the value 1; when awk is looking at the second record, NR has the value 2; and so on. In a BEGIN instruction, NR has the value 0. In an END instruction, NR contains the total number of records that were read. This instruction:

END { print NR }

prints the total number of data records read by the awk program.

FNR

Is like NR, but it counts the number of records that have been read so far *from the current file*. When you give several data files on the awk command line, awk sets FNR back to 1 when it begins reading each new file. Thus, a command such as:

{ printf "%d:%s\n",FNR,\$0 }

prints the line number in the current file, followed by a colon, followed by the contents of the current line.

NF

Gives the number of fields in the current record. For the **hobbies** file, NF is 4 for each line, because there are four fields in each record. In an arbitrary text file, NF gives the number of words on the current line in the file; by default, awk assumes that blanks separate the fields of a record, so it considers each word on a line to be a separate field. Therefore, the program:

{ count = count + NF }
END { print count }

prints the total number of words in the file.

Using these built-in variables, you can create more ambitious awk commands.

awk 'NF == 1 {print}' file

prints those records with precisely one field in them. There is no –F option specified for this command, so awk assumes that blanks or tab characters separate the fields. The foregoing command therefore prints all lines that contain only one word (that is, one field).

awk '{print FNR ": " \$0}' file

\$0 stands for the entire record. The foregoing command displays the contents of **file**, putting a line number and a colon before each line.

awk '/abc/ {print FILENAME ": " \$0}' *.bas

examines all files that have the **.bas** extension in the working directory. It prints every line that contains the string abc and also displays the filename, so you know which file contains which lines.

Built-in string variables

awk also provides a number of built-in string variables:

FILENAME

Contains the name of the current input file. For example, when running programs against the **hobbies** file, the value of FILENAME would be **hobbies** (if that is the file you are using). If the input is coming from the awk standard input, the value is -.

FS

Is the *field separator* string, giving the character that is used to separate fields in the current file. The default value for FS is "" (a single blank), which as a special case matches both blank and tab. However, if the command line contains an -F option specifying a different field separator, FS is a string containing the given separator character. 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 topic of the awk command description in <u>z/OS UNIX System</u> Services Command Reference.

RS

Is the *input record separator*. Just as FS indicates the character that separates fields within records, RS indicates the character that separates one record from another. By default, RS contains a newline character, which means that input records are separated by newlines. However, you can assign a different character to RS; for example, with:

RS = ";"

input records are separated by semicolons. This lets you have several records on a single line, or a single record that extends over several lines. Records are separated by a semicolon, not a <newline> character. As an important special case:

RS = ""

separates records by empty lines.

0FS

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 (numbers that are not integers) to strings as if you had specified the operation:

```
sprintf(CONVFMT, number ...)
```

By default, the value of CONVFMT is %.6g.

Note: CONVFMT is a POSIX extension not found in traditional implementations of awk.

Statements and loops

awk supports the following types of statements and loops:

- if statement
- while loop
- for loop

- next statement
- exit statement

The if statement

An if statement is an action of the form:

if (expression) statement1 else
statement2

Typically, the *expression* in the if statement has a true-or-false value. If the value is true, *statement1* is performed; otherwise, *statement2* is performed. The else *statement2* part is optional.

The while loop

A while loop repeats one or more other instructions as long as a given condition holds true. The format of the loop is:

while (*expression*) statement

where the statement can be a single statement or a compound statement.

The for loop

The statement:

```
for
 (expression1; expression2; expression3)
 statement
```

is equivalent to the following instruction sequence:

```
expression1
while (expression2) {
    statement
    expression3
}
```

The next statement

The next instruction skips immediately to the next record in the data file.

The exit statement

The exit statement makes an awk program behave as if it had just reached the end of data input. No further input is read. If there is an END action, awk executes it before the program ends. As with next, exit is often used when input data is found to be incorrect.

If exit appears inside the END action, the program ends immediately.

Functions

awk supports:

- Arithmetic functions
- String manipulation functions
- User-defined functions
- Passing an array to a function
- The getline function

Arithmetic functions

awk recognizes the most common mathematical functions, as shown in the following table.

Function	Result
sqrt(x)	Square root of <i>x</i>
sin(x)	Sine of <i>x,</i> where <i>x</i> is in radians
cos(x)	Cosine of <i>x</i> , where <i>x</i> is in radians
atan2(<i>y,x</i>)	Arctangent of y/x in range $-\pi$ to π
log(x)	Natural logarithm of x
exp(x)	The constant <i>e</i> to the power <i>x</i>
int(x)	Integer part of <i>x</i>
rand()	Random number between 0 and 1
srand(x)	Sets x as seed for rand()

Several of these functions may require more explanation.

The int function takes a floating-point number as an argument and returns an integer. The integer is just the floating-point number, without its fractional part.

Every call to rand returns a new random number between 0 and 1. In this way, you can get a sequence of random numbers. You can use srand to set the starting point, or seed, for a random number sequence. If you set the seed to a particular value, you always get the same sequence of numbers from rand. This is useful if you want a program to use rand but obtain uniform results every time the program runs.

String manipulation functions

awk has a number of functions that perform string operations:

length

Returns an integer that is the length of the current record (that is, the number of characters in the record, without the newline on the end). For example, the following program calculates the total number of characters in a file (except for newline characters):

```
{ sum = sum + length }
END { print sum }
```

length(s)

Returns an integer that is the length of the string s. For example, the following program prints the length of the first field in each record of the file:

```
{ print length($1) }
```

The function call length(\$0) is equivalent to just length.

gsub(regexp,replacement)

Puts the replacement string *replacement* in place of every string matching the regular expression *regexp* in the current record. For example, the program:

```
{
   gsub(/John/,"Jonathan")
   print
}
```

checks every record in the data file for the regular expression John, replaces matching strings with Jonathan, and prints the resulting record. As a result, the program's output is exactly like its input, except that every occurrence of John is changed to Jonathan. This form of the gsub function returns an integer telling how many substitutions were made in the current record. This is 0 if the record has no strings that match *regexp*.

sub(regexp,replacement)

Is similar to gsub, except that it replaces only the *first* occurrence of a string matching *regexp* in the current record.

gsub(regexp,replacement,string_var)

Puts the replacement string *replacement* in place of every string matching the regular expression *regexp* in the string *string_var*. For example, the program:

```
{
   gsub(/John/,"Jonathan",$1)
   print
}
```

is similar to the previous program, but the replacement is made only in the first field of each record. This form of the gsub function returns an integer telling how many substitutions were made in *string_var*.

sub(regexp,replacement,string_var)

Is similar to the previous version of gsub, except that it only replaces the *first* occurrence of a string matching *regexp* in the string *string_var*.

Note: You must use four backslashes to embed one literal backslash in a gsub() or sub() substitution string. For example,

```
gsub(/backslash/,"\\\\")
```

replaces all occurrences of the word backslash with the single character \.

index(string,substring)

Searches the given *string* for the appearance of the given *substring*. If it cannot find *substring*, index returns 0; otherwise, index returns the number (origin 1) of the character in *string* where *substring* begins. For example:

index("abcd","cd")

returns the integer 3 because cd is found beginning at the third character of abcd.

match(string,regexp)

Determines if *string* contains a substring that matches the regular expression (pattern) *regexp*. If so, the function returns an index giving the position of the matching substring within *string*; if not, match returns 0. match also sets a variable named RSTART to the index where the matching string starts, and a variable named RLENGTH to the length of the matching string.

substr(string,pos)

Returns the last part of *string*, beginning at a particular character position. The argument *pos* is an integer, giving the number of a character. Numbering begins at 1. For example, the value of:

substr("abcd",3)

is the string cd.

substr(string,pos,length)

Returns the part of *string* that begins at the character position given by *pos* and has the length given by *length*. For example, the value of:

```
substr("abcdefg",3,2)
```

is cd (a string of length 2 beginning at position 3).

sprintf(format,value1,value2,...)

Is based on the printf action. The value of sprintf is the string that would be printed out by the action

```
printf(format,value1,value2,...)
```

For example:

```
str = sprintf("%d %d!!!\n",2,3)
```

assigns the string "2 3!!!\n" to the string variable str.

tolower(string)

Returns the value of *string*, but with all the letters in lowercase. (This function is an extension to standard awk.)

toupper(string)

Returns the value of *string*, but with all the letters in uppercase. (This function is an extension to standard awk.)

ord(string)

Converts the first character of *string* into a number. This number gives the decimal value of the character in the character set used on the system. (This function is an extension to standard awk.)

User-defined functions

In an awk program, a function definition looks like this:

```
function name(argument-list) {
    statements
}
```

The *argument-list* is a list of one or more names (separated by commas) that represent argument values passed to the function. When an argument name is used in the *statements* of a function, it is replaced by a copy of the corresponding argument value.

For example, the following is a simple function that takes a single numeric argument *N* and returns a random integer between 1 and *N* (inclusive):

```
function random(N) {
    return (int(N * rand() + 1))
}
```

Passing an array to a function

When an array is passed as an argument to a function, it is passed by *reference*. This means that the function works with the actual array, not with a copy. Anything that the function does to the array has an effect on the original array. split is a built-in function that takes an array as an argument.

split(string,array)

split breaks up *string* into fields, and assigns each of the fields to an element of *array*. The first field is assigned to *array*[1], the next to *array*[2], and so on. Fields are assumed to be separated with the field separator string FS. If you want to use a different field separator string, you can use:

split(string,array,fsstring)

where *fsstring* is the field separator string you want to use instead of FS. The result of split is the number of fields that *string* contained.

Note: split actually changes the elements of array. When an array is passed to a function, the function may change the array elements.

The Getline function

The getline function reads input from the current data file or from a different file.

Running system commands

You can run commands with the system function:

system("command line")

runs the given command line: For example:

system("cd XYZ")

runs a cd command to change the working directory.

Controlling awk output

By default, awk output is written to your workstation screen. You can save the output of an awk program in a file by using *output redirection*. To do this, put:

>filename

on the end of any awk command line. For example:

awk -f progfile datafile >outfile

writes all the output from the awk program to a file named **outfile**. In this case, the output does not appear on the workstation screen.

Formatting the output

The output of the program:

```
$1 == "Jim" { print "$", $4/52 }
```

is:

\$ 1.92308
\$ 0.192308
\$ 1.34615

This output shows the amount of money per week that Jim spent on his hobbies. However, money amounts usually have only two digits after the decimal point. How can you change the program to make the money amounts appear more normal? The answer is to use the printf action instead of print. This lets you specify the *format* in which awk prints the output.

A printf action looks like this:

{ printf format-string, value, value, ... }

The *format-string* indicates the output format. The *values* are the data to be printed.

A format string contains two kinds of items:

- · Normal characters, which are just printed out as is
- Placeholders, which awk replaces with values given later in the printf action

As an example, try running the following program on the **hobbies** file:

\$2 == "bridge" { printf "%5s plays bridge\n", \$1 }

awk prints:

Jim plays bridge Linda plays bridge Lori plays bridge The format string:

"%5s plays bridge\n"

has one placeholder: %5s. When printf prints its output, replacing the placeholder with the value \$1, which is the first field of the record being examined. The rest of the format string is just printed out as is.

Note: The format string ends in \n; for more information, see "Escape sequences" on page 302.

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:

С

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).

е

A floating-point number in scientific notation, as in -d.ddd dddE+dd.

f

A floating-point number in conventional form, as in -ddd.ddd ddd.

g

A floating-point number in either e or f form, whichever is shorter; also, nonsignificant zeros are not printed.

0

S

A string.

Х

An unsigned integer in hexadecimal form (base 16).

An unsigned integer in octal form (base 8).

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
Escape	ASCII character
------------	----------------------------
\t	Horizontal tab
\v	Vertical tab
\000	ASCII character, octal ooo
\xdd	Hexadecimal value dd
\"	Quote
\ <i>c</i>	Any other character c

Appendix C. Code page conversion when the shell and MVS have different locales

A *code page* for a specific character set determines the graphic character produced for each hexadecimal encoding. The code page used is determined by the programs and national languages being used.

If the shell is using a locale generated with code pages IBM-1047, IBM-1027, or IBM-939, an application programmer needs to be concerned about variant characters in the POSIX portable character set whose encoding may vary from other EBCDIC code pages:

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

For example, the encodings for the square brackets do not match on code pages IBM-037 and IBM-1047:

- Left square bracket: [
- Right square bracket:]

Customizing the variant characters on your keyboard

Assuming that you are not using an APL character set, on many programmable workstations you can customize your keys so that you have hexadecimal encodings for the variant characters that match the shell-supported code pages. For example, for those brackets the compatible encodings would be:

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

Using the CONVERT option on the OMVS command

The OMVS command has a CONVERT option that lets you specify a conversion table for converting between code pages. The table you want to specify depends on the code pages you are using in MVS and in the shell. For example, if you are using code page IBM-037 in MVS and code page IBM-1047 in the shell, specify the following when you enter the OMVS command:

OMVS CONVERT((BPXFX111))

For more information, see the <u>OMVS</u> command description in <u>z/OS UNIX System Services Command</u> Reference.

When do you need to convert between code pages?

If you are using code page IBM-037 in MVS and the shell is using code page IBM-1047, you need to convert from one code page to another when:

- Transferring files between a workstation and the file system.
- Copying data between MVS data sets and the file system.
- Passing JCL pathname data to z/OS UNIX programs—unless you restrict yourself to characters in the POSIX portable file name character set.
- Passing JCL parameters and pathnames to a shell invoked from a batch program—unless you restrict yourself to characters in the POSIX portable file name character set.
- Converting between ASCII and EBCDIC when using the pax utility.

Methods for converting data

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

- To convert data you are typing at a 3270 terminal, you specify a conversion table *other than* BPXFX100 (the null conversion table) with the OMVS command. The data you type at your workstation when you are working in the shell is converted to a shell-supported code page.
- To convert data between code pages IBM-037 and IBM-1047 when you are moving the data to or from the hierarchical file system, you can use the CONVERT option on the OPUT, OGET, and OCOPY commands.
- To convert double-byte or single-byte data to a selected code page while you are working in MVS, use the z/OS XL C/C++ iconv utility. For information on how to use this utility, see <u>z/OS XL C/C++ User's</u> *Guide*.
- To convert double-byte or single-byte data to a selected code page while you are working in the shell, use the shell iconv utility.

The POSIX portable file name character set

To simplify conversion requirements, 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 (-)

The POSIX portable character set

The POSIX portable character set consists of

Uppercase A to Z Lowercase a to z Numbers 0 to 9

and these characters:

Characters			
+	<	=	>
\$	•	^	~
#	%	&	*
@	[1	١
{	}	I	!
п	1	()
,	-	-	•
1	:	;	?

Appendix D. Escape sequences for a 3270 keyboard

When using a 3270 keyboard, you can use escape sequences to type:

- Portable characters not included on your keyboard. See <u>"Escape sequences for portable characters not</u> on your keyboard" on page 309.
- Control characters that are normally available on ASCII workstations, but not EBCDIC ones. See "Escape sequences for control characters" on page 310.

The notation *EscChar* coupled with another letter (for example, <EscChar> m) indicates an escape sequence.

Escape sequences for portable characters not on your keyboard

If you do not have keys on your keyboard for the following portable characters, you can use an escape sequence to obtain them. Table 13 on page 309 lists the escape sequences for the portable characters.

Table 13: Portable characters: Escape sequences			
z/OS UNIX escape sequence	Character	ASCII control sequence	
<escchar> @ <escchar> 0</escchar></escchar>	<nul></nul>	Ctrl-@	
<escchar> g <escchar> G</escchar></escchar>	<alert></alert>	Ctrl-G	
<escchar> h <escchar> H</escchar></escchar>	<backspace></backspace>	Ctrl-H	
<escchar> i <escchar> I</escchar></escchar>	<tab></tab>	Ctrl-I	
<escchar> j <escchar> J</escchar></escchar>	<newline></newline>	Ctrl-J	
<escchar> k <escchar> K</escchar></escchar>	<vertical-tab></vertical-tab>	Ctrl-K	
<escchar> 1 <escchar> L</escchar></escchar>	<form-feed></form-feed>	Ctrl-L	
<escchar> m <escchar> M</escchar></escchar>	<carriage-return></carriage-return>	Ctrl-M	
<escchar> (</escchar>]	[
<escchar>)</escchar>]	1	

<tab> character: When you are writing makefiles for the make utility, you need to use a <tab> character. If you are using a shell editor, you can type a <tab> character as an <EscChar-I> sequence. After you press <Enter>, the tab displays as blank space.

If you are using the ISPF editor, you cannot type a <tab> character (ISPF handles only displayable characters).

Escape sequences for control characters

To obtain the control characters listed in Table 14 on page 310, you must use an escape sequence.

Table 14: Control characters: Escape sequences			
z/OS UNIX escape sequence	Character	ASCII control sequence	
<escchar> f <escchar> F</escchar></escchar>	<ack></ack>	Ctrl-F	
<escchar> x <escchar> X</escchar></escchar>	<can></can>	Ctrl-X	
<escchar> q <escchar> Q</escchar></escchar>	<dc1></dc1>	Ctrl-Q	
<escchar> r <escchar> R</escchar></escchar>	<dc2></dc2>	Ctrl-R	
<escchar> s <escchar> S</escchar></escchar>	<dc3></dc3>	Ctrl-S	
<escchar> t <escchar> T</escchar></escchar>	<dc4></dc4>	Ctrl-T	
<escchar> p <escchar> P</escchar></escchar>	<dle></dle>	Ctrl-P	
<escchar> y <escchar> Y</escchar></escchar>		Ctrl-Y	
<escchar> e <escchar> E</escchar></escchar>	<enq></enq>	Ctrl-E	
<escchar> d <escchar> D</escchar></escchar>	<eot></eot>	Ctrl-D	
<escchar> 2 <escchar> [</escchar></escchar>	<esc></esc>	Ctrl-[
<escchar> w <escchar> W</escchar></escchar>	<etb></etb>	Ctrl-W	
<escchar> c <escchar> C</escchar></escchar>	<etx></etx>	Ctrl-C	
<escchar> 6 <escchar> _</escchar></escchar>	<is1></is1>	Ctrl	

Table 14: Control characters: Escape sequences (continued)			
z/OS UNIX escape sequence	Character	ASCII control sequence	
<escchar> 5</escchar>	<is2></is2>	Ctrl-^	
<escchar> 4 <escchar>]</escchar></escchar>	<is3></is3>	Ctrl-]	
<escchar> 3 <escchar></escchar></escchar>	<is4></is4>	Ctrl-\	
<escchar> u <escchar> U</escchar></escchar>	<nak></nak>	Ctrl-U	
<escchar> o <escchar> O</escchar></escchar>	<si></si>	Ctrl-O	
<escchar> n <escchar> N</escchar></escchar>	<\$0>	Ctrl-N	
<escchar> a <escchar> A</escchar></escchar>	<soh></soh>	Ctrl-A	
<escchar> b <escchar> B</escchar></escchar>	<stx></stx>	Ctrl-B	
<escchar> z <escchar> Z</escchar></escchar>		Ctrl-Z	
<escchar> v <escchar> V</escchar></escchar>	<syn></syn>	Ctrl-V	

Escape sequences unique to a conversion table

Depending on the conversion table that you specify with the CONVERT keyword on the OMVS command, you may need to type a unique escape sequence to enter a character. This information shows how unique escape sequences are translated by each of the character conversion tables. The translations for escaped alphabetic characters (which are the same for all tables—these are Ctrl-A through Ctrl-Z) are not shown in these tables.

BPXFX100 conversion table

Table 15 on page 311 shows the escape sequences for certain characters that may not be on your keyboard.

Table 15: Translation of selected escaped characters (BPXFX100)			
z/OS UNIX escape sequence	Character	ASCII control sequence	
<escchar> ? <escchar> # <escchar> 7</escchar></escchar></escchar>		Ctrl-?	

Table 15: Translation of selected escaped characters (BPXFX100) (continued)			
z/OS UNIX escape sequence	SUNIX escape sequence Character ASCII control sequence		
<escchar> {</escchar>	[[
1]		
<escchar> ¬</escchar>	<is2></is2>	Ctrl-^	

BPXFX111 and BPXFX211 conversion tables

Table 16 on page 312 shows the escape sequences for certain characters that might not be on your keyboard.

Table 16: Translation of selected escaped characters (BPXFX111 and BPXFX211)			
z/OS UNIX escape sequence	Character	ASCII control sequence	
<escchar> ? <escchar> # <escchar> 7</escchar></escchar></escchar>		Ctrl-?	
<escchar> {</escchar>	[[
]]		
<escchar> ¬</escchar>	<is2></is2>	Ctrl-^	

BPXFX437, BPXFX450, BPXFX471, BPXFX473, BPXFX477, BPXFX478, BPXFX480, BPXFX484, BPXFX485, BPXFX497 conversion tables

Conversion tables BPXFX437, BPXFX450, BPXFX471, BPXFX473, BPXFX477, BPXFX478, BPXFX480, BPXFX484, BPXFX485, and BPXFX497 have the escape sequences listed in <u>Table 17 on page 312</u> for certain characters that might not be on your keyboard.

Table 17: Translation of selected escaped characters (BPXFX437, BPXFX450, BPXFX471, BPXFX473, BPXFX477, BPXFX478, BPXFX480, BPXFX484, BPXFX485, and BPXFX497). (BPXFX437, BPXFX450, BPXFX471, BPXFX473, BPXFX477, BPXFX478, BPXFX480, BPXFX484, BPXFX485, and BPXFX497)

z/OS UNIX escape sequence	Character	ASCII control sequence
<escchar> ? <escchar> 7</escchar></escchar>		Ctrl-?
<escchar> -</escchar>	~	
<escchar> %</escchar>	@	
<escchar> &</escchar>	\$	
<escchar> ; <escchar> !</escchar></escchar>	1	
<escchar> '</escchar>	^	
<escchar> =</escchar>	#	
<escchar> "</escchar>	x	

Table 17: Translation of selected escaped characters (BPXFX437, BPXFX450, BPXFX471, BPXFX473, BPXFX477, BPXFX478, BPXFX480, BPXFX484, BPXFX485, and BPXFX497). (BPXFX437, BPXFX450, BPXFX471, BPXFX473, BPXFX477, BPXFX478, BPXFX480, BPXFX484, BPXFX485, and BPXFX497) (continued)

z/OS UNIX escape sequence	Character	ASCII control sequence
<escchar> /</escchar>	\	
<escchar> :</escchar>	!	
<escchar> <</escchar>	{	
<escchar> ></escchar>	}	
<escchar> ^</escchar>	<is2></is2>	Ctrl-^

Appendix E. Locale objects, source files, and charmaps

The z/OS shells and utilities support the locales listed in the appendix in *z/OS XL C/C++ Programming Guide*.

A *locale name* is the same as a *locale object name*. The suffix of the locale name, for example, IBM-277, indicates the code page that the locale is based on.

The *symbolic link* is a shortened name for the complete locale object name; You can use the *symbolic link* name when specifying a locale for an environment variable or with the setlocale() function. For example, you can specify

LANG=En_US

instead of

LANG=En_US.IBM-1047

The compiled locale object files are in the directory **/usr/lib/nls/locale**. The locale source definition files are in **/usr/lib/nls/localedef**. The source file name combined with the code page name results in the name of the locale object.

The charmap files are in **/usr/lib/nls/charmap**. The charmap file names are identical to code page names — for example, IBM-1047.

Appendix F. Accessibility

Accessible publications for this product are offered through <u>IBM Knowledge Center (www.ibm.com/</u> support/knowledgecenter/SSLTBW/welcome).

If you experience difficulty with the accessibility of any z/OS information, send a detailed email message to mhvrcfs@us.ibm.com.

Accessibility features

Accessibility features help users who have physical disabilities such as restricted mobility or limited vision use software products successfully. The accessibility features in z/OS can help users do the following tasks:

- Run assistive technology such as screen readers and screen magnifier software.
- Operate specific or equivalent features by using the keyboard.
- Customize display attributes such as color, contrast, and font size.

Consult assistive technologies

Assistive technology products such as screen readers function with the user interfaces found in z/OS. Consult the product information for the specific assistive technology product that is used to access z/OS interfaces.

Keyboard navigation of the user interface

You can access z/OS user interfaces with TSO/E or ISPF. The following information describes how to use TSO/E and ISPF, including the use of keyboard shortcuts and function keys (PF keys). Each guide includes the default settings for the PF keys.

- z/OS TSO/E Primer
- z/OS TSO/E User's Guide
- z/OS ISPF User's Guide Vol I

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users who access IBM Knowledge Center with a screen reader. In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), they can appear on the same line because they are considered a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that the screen reader is set to read out punctuation. All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The * symbol is placed next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is given the format 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol to provide information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, it indicates a reference that is defined elsewhere. The string that follows the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you must refer to separate syntax fragment OP1.

The following symbols are used next to the dotted decimal numbers.

? indicates an optional syntax element

The question mark (?) symbol indicates an optional syntax element. A dotted decimal number followed by the question mark symbol (?) indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element, (for example 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that the syntax elements NOTIFY and UPDATE are optional. That is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.

! indicates a default syntax element

The exclamation mark (!) symbol indicates a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicate that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the dotted decimal number can specify the ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In the example, if you include the FILE keyword, but do not specify an option, the default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, the default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP applies only to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

* indicates an optional syntax element that is repeatable

The asterisk or glyph (*) symbol indicates a syntax element that can be repeated zero or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data area, you know that you can include one data area, more than one data area, or no data area. If you hear the lines 3* , 3 HOST, 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

Notes:

- 1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
- 2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you can write HOST_STATE, but you cannot write HOST_HOST.

3. The * symbol is equivalent to a loopback line in a railroad syntax diagram.

+ indicates a syntax element that must be included

The plus (+) symbol indicates a syntax element that must be included at least once. A dotted decimal number followed by the + symbol indicates that the syntax element must be included one or more times. That is, it must be included at least once and can be repeated. For example, if you hear the line 6.1+ data area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. Similar to the * symbol, the + symbol can repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loopback line in a railroad syntax diagram.

Notices

This information was developed for products and services that are offered in the USA or elsewhere.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing IBM Corporation North Castle Drive, MD-NC119 Armonk, NY 10504-1785 United States of America

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing Legal and Intellectual Property Law IBM Japan Ltd. 19-21, Nihonbashi-Hakozakicho, Chuo-ku Tokyo 103-8510, 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.

This information could include missing, incorrect, or broken hyperlinks. Hyperlinks are maintained in only the HTML plug-in output for the Knowledge Centers. Use of hyperlinks in other output formats of this information is at your own risk.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites 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 Site Counsel 2455 South Road 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 measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or

reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM Online Privacy Statement

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

Depending upon the configurations deployed, this Software Offering may use session cookies that collect each user's name, email address, phone number, or other personally identifiable information for purposes of enhanced user usability and single sign-on configuration. These cookies can be disabled, but disabling them will also eliminate the functionality they enable.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at <u>ibm.com/privacy</u> and IBM's Online Privacy Statement at <u>ibm.com/privacy/details</u> in the section entitled "Cookies, Web Beacons and Other Technologies," and the "IBM Software Products and Software-as-a-Service Privacy Statement" at ibm.com/software/info/product-privacy.

Policy for unsupported hardware

Various z/OS elements, such as DFSMS, JES2, JES3, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see: <u>IBM Lifecycle Support for z/OS (www.ibm.com/</u> software/support/systemsz/lifecycle)
- For information about currently-supported IBM hardware, contact your IBM representative.

Programming interfaces

This publication primarily documents information that is NOT intended to be used as Programming Interfaces of z/OS UNIX System Services.

This publication also documents intended Programming Interfaces that allow the customer to write programs to obtain the services of z/OS UNIX System Services. This information is identified where it occurs, either by an introductory statement to a chapter or section or by the following marking:

[Programming Interface Information] [End Programming Interface Information]

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (° or [™]), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at <u>Copyright and Trademark information</u> (www.ibm.com/legal/copytrade.shtml).

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Acknowledgments

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

Index

Special Characters

_BPX_BATCH_SPAWN environment variable 149 _BPX_BATCH_UMASK environment variable 149 _BPX_SHAREAS variable <u>161</u> _BPX_SPAWN_SCRIPT variable 40 -- option 61, 85 ; (semicolon) 68, 91 ?74,97 . (dot) 192 . (dot) shell command 108 .. (dot dot) 192 .profile file 35 ' ' escape character 73, 96 " " escape character 74 [[]] double square brackets 119 * 74, 96 *** prompt 16 /dev directory 179 /dev/null 64, 88 /etc/profile 35 \ continuation character 18, 32 \ escape character 73, 95 && 68,91 #! 108, 126 • • syntax 69, 92 < 88 > 63, 87 > prompt 19, 73, 96 >> 63, 87 || 68, 91 \$ prompt 13 \$-117 \$?117 \$() syntax 69, 92 \$@ 117 \$* 117 \$#117 \$N construct 112, 129

Numerics

2> <u>64</u>, <u>88</u>
3270 emulation <u>11</u>
3270 pass-through mode keyword on the OMVS command 25

A

access ACL 225 access control list (ACL) using 225 accessibility contact IBM 317 features 317 action print 285 action (continued) printf 300 address socket 179 TCP/IP X-Window application 4 address alias 164 address space limit for kernel 161 shared keyword on OMVS command 26 TSO/E working directory 185 ADSTAR 211 alarm keyword on OMVS command 24 alias address 164 defining 65, 88 mailx 164 redefining 66, 89 tracking 67 turning off 68, 90 alias shell command 65, 88 ALLOCATE TSO/E command example 255, 260, 264 path name and data set name requirement 182 specifying standard files 63, 87 appending to an archive 215 application hung 28 archive file copying into a file system 272 installing into the file system 270 transferring to a data set 271 transferring to tape or diskette 272 archive viewing 214 argument array 299 arithmetic calculation 109, 127 function 297 operator 287 array, used in awk 299 ASCII terminal interface 31 assistive technologies 317 audit file 222 autoloading 123 autoscrolling keyword on OMVS command 24 awk utility blanks and horizontal tabs 283 command line 285 compound assignment 289 controlling output 300 data files 281 escape sequences 302 formatting output 300 functions 296

awk utility (continued) output 285, 300 print action 285 printf action 300 program shape 282 running a program 285 running system commands 300

В

background job canceling 138 exiting the shell 28 moving to foreground 136 suspended 136 TSO/E 135 using BPXBATCH or & 140 backing up files backing up a directory 213 from the shell 212 manually 211 selected 211 selected by date 216 system 211 backslash (\) character 73, 95 backup file TSM 211 base ACL entry 225 batch job BPXBATA2 and BPXBATA8 146 BPXBATCH 140, 145 **BPXBATSL 146** support for path name 143 **BEGIN pattern 292** bit bucket 64, 88 SETGID 225 SETUID 225 sticky 220, 222 blank screen clearing with a form-feed 24 **BPX.SUPERUSER FACILITY 80, 103 BPXBATA2**, alias for BPXBATCH 146 BPXBATA8, alias for BPXBATCH 146 **BPXBATCH** environment variable file (STDENV) 147 invoked from TSO/E 154 invoked in the OSHELL REXX exec 155 invoked with JCL running a shell command 153 running a shell script 152 running an executable file or REXX exec 154 national language support 42, 55 parameter file (STDPARM) 149 **REGION size 152** running a background job 140 standard input, output and error 146 STDENV 147 STDPARM 149 STEPLIB data sets, cataloged 152 **BPXBATSL**, alias for **BPXBATCH 146** BPXFX100 escape sequences 311 BPXFX111

BPXFX111 (continued) escape sequences 312 BPXFX211 escape sequences 312 BPXFX311 254, 263 BPXFX437 escape sequences 312 BPXFX450 escape sequences 312 BPXFX471 escape sequences 312 BPXFX473 escape sequences 312 BPXFX477 escape sequences 312 BPXFX478 escape sequences 312 BPXFX480 escape sequences 312 BPXFX484 escape sequences 312 BPXFX485 escape sequences 312 BPXFX497 312 bpxtrace job tracing 136 bracket character code page conversion 305 BSAM access, HFS files 144 buffer size, output keyword on the OMVS command 25 built-in variable numeric 293 string 294 byte-range locking 211

С

cancel shell command 249 canonical mode 14 carriage return 20 case-sensitive processing 200 cat shell command 210 catalog master 179 user 179 cd shell command 192 changing a password or password phrase 80, 103 character conversion table keyword on OMVS command 24 character set double-byte using 28 doublebyte using 32 portable file name 199, 306 POSIX portable file name 306 character special file 179 characters variant 4, 43, 56, 305 chaudit shell command 222 chgrp shell command 224 child process 135 chmod shell command 219

chown shell command 224 cksum shell command 79, 102 CLIST 8 code page conversion copying data 266 DBCS data 266 doublebyte data 306 iconv command 267 OMVS command CONVERT option 305 square brackets 305 UUCP commands 169 with Network File System 186 code set 266 code set conversion automatic 190 combined commands filter 69, 92 pipe 69, 92 command argument 4, 62, 86 combining more than one 68, 91 continuation character () 18, 32 delaying execution 139 editing 77, 99 file system shell 184 flag 4 history function keys 77, 99 r command 76, 98 interrupting 19, 32 option 4, 61, 85 retrieving 75, 98 running after logoff 139 substitution 69, 92 usage 62 usage help 86 command line awk 285 editing 77, 99 hiding keyword, OMVS command 24 **Communications Server session** ISPF Edit 32 multiple logins 32 comparison operator 287 compound assignment 289 compress shell command running in batch 153 console file 179 construct using quotes around 113, 130 contact z/OS 317 continuation character (\) 18, 32 prompt 73, 96 control messages and online conversations 168 control characters escape sequences 310 Control function key using a 19

control structure for loop 121, 134 if conditional 119, 132 while loop 121, 133 control structures combining 122 conversion between code pages 214, 266 OMVS command CONVERT option 305 table copy commands 266 CONVERT copy command conversion tables 266 converting files between 189 copy data set into a data set OCOPY command 263 data set into a directory **OPUTX command 256** data set into a file cp command 253, 256 OCOPY command 254 **OPUT command 253** data using TSO/E commands 251 data using z/OS shell commands 251 DBCS data 267 directory into a data set cp command 261 OGETX command 261 executable into a file cp command 264 file into a data set OCOPY command 259 OGET command 258 file into a file cp command 262 OCOPY command 263 pax command 263 file into data set cp command 258 load module into a data set 264 load module into a file 264 MBCS data 267 VSAM data set 257 cp command 253, 256, 258, 261, 264 cp shell command default permissions 218 cron daemon 4 Ctrl-C 32 current working directory 191 customization .tcshrc 51 ENV variable 38 kevboard 20 OMVS command 23 PATH variable 38, 52 profile file 35 shell interface 23 shell options 28, 46, 59 square brackets 305 tcsh shell startup files 49

D

daemons 4 data access 180 data set allocating 254 cataloged 45, 58 copying cp command 253, 256 into a file 252, 256, 263 load module into a file 264 OCOPY command 254, 263 **OPUT command 253 OPUTX command 256** deleting 270 executable module copying 264 hierarchical file system (HFS) 178 load module copying 264 STEPLIB cataloging 45, 58 DD statement in JCL ddnames 143 pathname keywords 143 z/OS UNIX support 143 ddname 143 debug data wrapping keyword on OMVS command 26 debugging keyword on the OMVS command 24 decrement operator 289 DELETE TSO/E command 270 dev directory 179 DFSMS management of HFS data sets 178 DFSMS/MVS Network File System feature 186 DFSMShsm data set backup and restore 211 HFS data set back up and restore 178 diff shell command 196, 205 directory access using ACLs 225 changing 192 comparing contents 196 copying cp command 261 OGETX command 261 creating 193 default permissions 193, 218 dev 179 finding 197 listing contents 195 permissions default 218 displaying 223 removing 195 specifying name 191 sticky bit 220, 222 working 191 directory default ACL 225

displaying a user name 81, 104 Distributed File System (DFS) 178 distribution list sending a message to 173 dot notation 192 double quotatioin marks enclosing a construct 113 double quotation marks enclosing a construct 74, 96, 130 double square brackets 119 double-byte character set alias names 65 exporting a variable name 65 keyword on OMVS command 24 using 32 using a 28 double-byte data code page conversion 306 DSNTYPE keyword 144 dump nontext file 65, 88 dynamic link library (DLL) environment variable 40, 54

E

echo shell command 37, 51 ed editor default permissions 218 using 239 editor command editing 77, 99 ed 239 **ISPF 227** sed 245 vi 228 effective group ID 225 effective user ID 225 emacs editor 78, 100 emulation 3270 11 END pattern 292 Enhanced ASCII file tagging 189 porting 190 ENV variable setting 38 environment file 38, 51 environment variable BPX_BATCH_SPAWN 149 BPX_BATCH_UMASK 149 **BPX SHAREAS 161** BPX_SPAWN_SCRIPT 40 changing dynamically 37, 50 displaying 37, 51 ENV. setting 38 file (STDENV) 147 LANG 41, 44, 55, 57 LC_ALL 41, 55 LC_COLLATE 41, 55 LC_CTYPE 41, 55 LC_MESSAGES 41 LC_SYNTAX 43, 56 LOCPATH 44, 57 PATH setting 38, 52 STEPLIB 45, 58

environment variable (continued) **TMP VI 238** TZ 44, 57 error redirection 64, 88 standard 62, 86 error message shell 14 escape character keyword on OMVS command 25 notation 18 shell command 73, 95 sequence BPXFX100 table 311 BPXFX111 table 312 BPXFX211 table 312 BPXFX437 table 312 BPXFX450 table 312 BPXFX471 table 312 BPXFX473 table 312 BPXFX477 table 312 BPXFX478 table 312 BPXFX480 table 312 BPXFX484 table 312 BPXFX485 table 312 BPXFX497 table 312 control characters 310 portable characters 309 tables 309 escape sequences 312 EscChar notation 18 EscChar-C 19, 28 EscChar-D 20. 27 EscChar-V 28 EscChar-Z 139 etc/profile 35 exec shell command 64 executable copying cp command 264 executable file invoked with BPXBATCH and JCL 154 executable module copying into a data set 264 copying into the file system 264 exit shell command 27 exit statement 296 expansion preventing wildcard 46, 59 export shell command 110 export variable 36, 46, 110, 128 expressions 109, 127 extattr shell command 181 extended ACL entry 225 external link deleting 205 DLL support 204 locale object files 204 NFS client support 204 sticky bit 183

F

feedback xxi field 282 FIFO special file 179 file .tcshrc 51 access auditing 222 BSAM, QSAM 144 program 211 using ACLs 225 allocating 254 analyzing contents 208 awk program 286 back up and restore 211 browsing 210, 211 changing ownership 224 closing 64 comparing two 205 copying cp command 258, 262 OCOPY command 259, 263 OGET command 258 pax command 263 creation mode mask 224 default permissions ed 244 ISPF Edit 227 OEDIT 227 deleting 201 descriptor 63 displaying contents 210, 211 editing with ISPF 227 environment variables for BPXBATCH 147 erasing 201 executable 143 finding 197 formatted browsing 211 formatting 247 I/O 180 inode number 202 locking 211 login script 38 moving 205 naming 199 nontext dumping 65, 88 opening with JCL 144 parameter string for BPXBATCH (STDPARM) 149 permissions default 218 displaying 223 printing 247 profile file example 35 removing 222 renaming 205, 222 searching pattern 209 string 208 sending 167 sh history 76, 98 sorting contents

file (continued) sorting contents (continued) example 207 sticky bit 220 transfer to a workstation 270 to the host 270 **UUCP 169** file default ACL 225 file descriptor file 179 file name creating 199 length 199 listing 197 portable file name character set 199 using a wildcard character 74, 96 file name completion using 100 file system data access 180 I/O 180 mountable 177 permissions 217 root 177 security 217 shell commands 184 using the ISPF shell 157, 186 file tagging Enhanced ASCII 189 Unicode Services 190 file/etc/profile 35 filter 69, 92 find shell command 70, 79, 86, 93, 102, 197 flag. shell command 4 FOMTLINP module 31 fopen() function 144 for loop 121, 134, 296 foreground job canceling 138 moving to background 136 form-feed character 24 formatting files pr command 247 fsck shell command 186 FSUM messages 83, 105 FTAM function **OSI/File Services 270** ftp 27 function arithmetic 297 getline 299 passing an array to 299 string manipulation 297 user-defined 299 using 122 function key customizing keyword on OMVS command 25 description of function 15 display keyword on OMVS command 25 displaying the settings 13 setting keyword on OMVS command 25

fuser utility 216

G

getline function <u>299</u> GID changing <u>224</u> Greenwich Mean Time (GMT) <u>44, 57</u> grep shell command 66, 90, 208

Η

hard link deleting 205 head shell command 210 help facility 105 HELP TSO/E command 185 HFS data set backing up and restoring 178 power failure 186 history file editing commands 76, 99 history shell command 75, 76, 98 hung application 28

I

iconv shell command example 267 iconv utility z/OS XL C/C++ 266, 306 identifier job 135 process 135 IEWBLINK copying executables to file 266 copying load module to file 265 if conditional 119, 132 if statement 296 **IKJETF01 256** increment operator 289 inetd daemon 4 inode number 202 input redirection 64, 88 standard 62, 86 **INPUT HIDDEN indicator 21 INPUT** indicator 20 Interactive System Productivity Facility 200 ISPF browsing a file 211 case-sensitive processing 200 editing a file sequence numbers 148 ISPF command 16 NUMBER OFF 148 sequence numbers 148 shell help facility 158 locale 44, 57 uppercase processing 200 ISPF TSO/E command 16

J

JCL case-sensitive processing 200 ddnames 143 example using OCOPY 255, 260, 264 path name and data set name requirement 182 path name support 143 shell commands 7 specifying standard files 63, 87 submitting 145 JES printer 247 job background canceling 138 moving to foreground 136 stopping 139 suspended 136 control commands 135 foreground canceling 138 moving to background 136 stopping 139 identifier 135 priority 135 resuming stopped 139 status 137 tracing 136 job control language 7 job entry subsystem 247 jobs shell command 137

Κ

keyboard escape sequence BPXFX100 table 311 BPXFX111 table 312 BPXFX211 table 312 BPXFX437 table 312 BPXFX450 table 312 BPXFX471 table 312 BPXFX473 table 312 BPXFX477 table 312 BPXFX478 table 312 BPXFX480 table 312 BPXFX484 table 312 BPXFX485 table 312 BPXFX497 table 312 tables 309 navigation 317 PF keys 317 remapping 20 shortcut keys 317 kill shell command 135, 138 Korn shell 3

L

LANG variable <u>41</u>, <u>44</u>, <u>55</u>, <u>57</u> language of messages <u>44</u>, <u>57</u> LC_ALL variable <u>41</u>, <u>55</u>

LC CTYPE variable 41, 55 LC_MESSAGES 55 LC MESSAGES variable 41, 55 LC SYNTAX variable limitations 44, 57 lex shell command locale modifications 41, 54 LIBPATH variable 40, 54 line mode 14 LINES keyword, OMVS command 25 link external 179, 186, 204 hard 202 symbolic 179, 202 In shell command 202 load module copying into a data set 264 copying into a z/OS UNIX file 264 locale changing 54 code page conversion 305 customizing lex, mailx, make, and yacc 41, 54 default 4 ISPF shell 44, 57 LC_SYNTAX example 43, 56 limitations 44, 57 lex, mailx, make, and yacc 41, 54 LOCPATH variable 44, 57 object files 44, 57 REXX execs 44, 57 selecting 41 selecting a <u>43, 54, 56</u> shell and utilities, changing 54 variant characters 4, 43, 56, 305 locale name 315 locale object files 204 LOCPATH variable 44, 57 login from a remote system 31 multiple 32 name 193 script 38, 51 logout shell 27 loop for 296 while 296 lp shell command 248 lpstat shell command 249 ls command for displaying file information 183 ls shell command 195, 223

LC COLLATE variable 41, 55

Μ

magic number <u>108</u>, <u>126</u> mail, steps for sending <u>163</u> mailx shell command locale modifications <u>41</u>, <u>54</u> make shell command locale modifications <u>41</u>, <u>54</u> man shell command 82, <u>105</u>

mask file creation mode 224 master catalog 179 matching operator 289 member partitioned data set naming requirements 261 mesg shell command 168 messages broadcasting 168 controlling 168 language of 44, 57 receiving 164, 173 sending to MVS operator 164, 173 shell 83, 105 vi/ex file recovered 237 metacharacter 70, 93, 209 mkdir shell command default permissions 218 MKDIR TSO/E command default permissions 218 mode cp command 218 default directory 193 directory creation 218 file creation 218, 227 ed command 218 mask file creation 224 mkdir command 218 MKDIR command 218 OCOPY command 218 oedit command 218 **OEDIT command 218 OPUT command 218** redirection creating a file 218 vi command 218 modified expansion 114, 131 more shell command 210 MORE... indicator 20 mountable file system 177 multiple commands filter 69, 92 pipe 69, 92 multiple logins 32 multiple sessions asynchronous terminal interface 32 keyword on OMVS command 26 **OPEN** subcommand 17 switching between 16 multiple-condition operator 289 mv shell command 205, 222, 251 MVS operator sending a message to 164, 173

Ν

name file <u>199</u> login <u>193</u> named pipe 179 navigation keyboard 317 nawk utility 281 Network File System feature code page conversion 186 external link 186 running an NFS-mounted executable 269 newline character appending 20 suppressing 20 next statement 296 NEXTSESS subcommand 16 nice shell command 135 nohup shell command z/OS shell processing 140 NOT ACCEPTED indicator 21 NOT ACCEPTED/MORE indicator 21 notation dot 192 tilde (~) 192 null file 179 numeric value 284 numeric variable, built-in 293

0

obrowse shell command 210 **OBROWSE TSO/E command** path name and data set name requirement 182 OCOPY TSO/E command default permissions 218 octal numbers 220 od shell command 65, 88 oedit shell command default permissions 218 OEDIT TSO/E command default permissions 218 path name and data set name requirement 182 OGET TSO/E command path name and data set name requirement 182 OGETX TSO/E command 261 OMVS TSO/E command CONVERT option 305 customizing 23 invoking the shell 11 subcommands 22 online conversation having 167 online help 105 **OPEN** macro 144 **OPEN** subcommand 17 operation compound assignment 289 ordering 288 operator arithmetic 287 comparison 287 increment or decrement 289 matching 289 multiple-condition 289 operator message sending 164, 173 option settings shell session deletion verification 60

option settings (continued) shell session (continued) displaying 47, 59 option, shell command 85 **OPUT TSO/E command** default permissions 218 path name and data set name requirement 182 **OPUTX TSO/E command 256** order, arithmetic operation 288 **OS/2** Extended Edition SEND and RECEIVE programs 269 OSHELL REXX exec 19, 155 **OSI/File Services** FTAM function 270 output awk controlling 300 redirection 63, 87 standard 62, 86 output buffer size keyword on the OMVS command 25

Ρ

parameter expansion 114, 131 positional 114, 131 special 116, 131 parameter string for BPXBATCH file 149 parent process 135 partitioned data set member names 261 pass-through mode, 3270 keyword on the OMVS command 25 passwd shell command 80, 103 password or password phrase changing 80, 103 path 181 PATH keyword 143 path name JCL requirement 182 resolving symbolic links 182 TSO command requirement 182 PATH variable setting 38, 52 PATHDISP keyword 143 PATHMODE keyword 143 pathname JCL 143 PATHOPTS keyword 143 pattern matching 209 pattern, awk ranges 292 simple 282 special 292 pax (copy mode) shell command 263 PC 3270 emulation program SEND and RECEIVE programs 269 performance shared address space 161shell script 40 permissions bits 217 changing 219 cp command 218

permissions (continued) default directory 193 directory creation 218 file creation 218 ISPF Edit 227 OEDIT 227 summary 218 displaying 223 ed command 218 mkdir command 218 MKDIR command 218 OCOPY command 218 octal 220 oedit command 218 OEDIT command 218 OPUT command 218 redirection creating a file 218 symbolic 219 vi command 218 PF key 13 pg shell command 210 PGID 135 PID 135 pipe named 179 unnamed 179 pipeline 69, 92 placeholders 301 portable characters escape sequences for 309 portable file name character set 306 porting considerations 190 positional parameter 112, 114, 129, 131 POSIX portable file name character set 306 power failure 186 PPID 135 pr shell command 211, 247 PREVSESS subcommand 17 print action, awk utility 285 PRINTDS TSO/E command 249 printenv shell command 37, 51 printf action, awk utility 300 printing checking job status 249 lp command 248 TSO/E commands 248 z/OS Print Server 248 process child 135, 161 ending 135 group 135 identifier 135 limit per user 161 parent 135, 161 priority 135 process IDs, listing 216 PROFILE PLANGUAGE TSO/E command 28 profile.profile 35 profile/etc/profile 35 program awk, running 285 file, awk 286

program (continued) timing 80, 102 program function key 13 programming 85 prompt *** 16 prompt, continuation 73, 96 ps shell command 137 pwd shell command 191

Q

QSAM access, HFS files <u>144</u> QUIT subcommand <u>17</u> quotation marks enclosing a construct <u>113</u> quotes enclosing a construct <u>130</u>

R

r shell command 76 RACF **BPX.SUPERUSER FACILITY 80, 103** random number files 179 ranges, in a pattern 292 **RECEIVE program 269 RECEIVE TSO/E command 172** record keeping 79, 101 records 281 redirection controlling 46, 59 creating a file default permissions 218 **REGION size, BPXBATCH 152** regular expression 210, 290 regular file 179 relative pathname dot notation 192 tilde notation 192 remap keyboard 20 remote login 31 rename shell command 222 renice shell command 135 Resource Access Control Facility 4 restoring files file system 211 from the shell 212, 216 restoring a directory 213 retrieve function key 77, 99 retrieving commands 75, 98 return statement 123 REXX calling z/OS UNIX System Services 8 **OSHELL 155** z/OS UNIX extensions 156 rlogin 31 rlogin session ISPF Edit 32 multiple logins 32 retrieving commands 77, 99 rlogin shell command, porting 31 rm shell command 67, 90, 195, 201, 222 rmdir shell command 195, 222 root directory 177 **RUNNING indicator 20**

S

screen clearing with a form-feed 24 SDSF (System Display and Search Facility) print job 247 search path verifying 39, 53 searching files 208 security RACF 4 sed editor using 245 SEND program 269 SEND TSO/E command 172 sending a file 167 sending a message 163, 166, 172 sending mail, steps for 163 sending to IBM reader comments xxi sequence numbers, ISPF 148 sessions ASCII terminal limitations 32 keyword on OMVS command 26 using multiple shell 22 set shell command 28, 37, 46, 51, 59 set-group-ID bit 225 set-user-ID bit 225 setlocale() 44, 57, 204 sh history file 76, 98 shared address space keyword on OMVS command 26 shell changing the locale 40 command escape characters 73, 95 invoked with BPXBATCH 154 invoked with BPXBATCH and JCL 153 run from TSO/E 155 command -- option 61, 85 daemons 4 differences from UNIX or AIX 11 entering TSO/E commands 26 error message 14 escape sequence BPXFX100 table 311 BPXFX111 table 312 BPXFX211 table 312 BPXFX437 table 312 BPXFX450 table 312 BPXFX471 table 312 BPXFX473 table 312 BPXFX477 table 312 BPXFX478 table 312 BPXFX480 table 312 BPXFX484 table 312 BPXFX485 table 312 BPXFX497 table 312 tables 309 exiting using NOHUP 140 with a background job 140 with a nohup background job 139 function 122

shell (continued) invoking 11 ISPF help facility 158 login 11 logout 27 messages 83, 105 metacharacter 70, 93 options deletion verification 60 displaying settings 47, 59 setting 28, 46, 59 prompt default 13 remote login 31 screen description 13 script executable 107, 125 function 122 invoked with JCL using BPXBATCH 152 running 107, 125 special characters 70, 93 special parameters 116, 131 using multiple sessions 22 variable arithmetic calculation 109, 127 creating 108, 126 exporting 36, 46, 110, 128 z/OS UNIX locale 44, 57 shell command alias 88 awk 281 cat 210 cd 192 chaudit 222 chgrp 224 chmod 219 chown 224 cksum 79, 102 compress 153 cp 262 diff 196, 205 echo 37, 51 exec 64 exit 27 export 110 extattr 181 find 70, 79, 86, 93, 102, 197 fsck 186 grep 66, 90, 208 head 210 history 75, 76, 98 iconv 266, 267, 306 jobs 137 kill 138 ln 202 lp 248 ls 195, 223 mailx 163 man 82, 105 mesg 168 mkdir 193 more 210 mv 205, 222 nice 135

shell command (continued) nohup 139 obrowse 210 od 65, 88 options 61 passwd 80, 103 pax (copy mode) 263 pg 210 pr 211, 247 printenv 37, 51 ps 137 pwd 191 r 76 rename 222 renice 135 rm 67, 90, 195, 201, 222 rmdir 195, 222 set 28, 37, 46, 51, 59 sort 206 stty 136 su 80, 103 submit 145 tail 210 talk 167 test 117 time 80, 102 tso 26, 81, 104, 108, 126 typeset 111 umask 224 uucp 169 uulog 171 uupick 171 uustat 171 uuto 169 uux 172 wait 139 wall 168 wc 208 whence 39 which 53 whoami 81, 104 writing 166 shell script performance improving 40 skulker 201 writing 107 shortcut keys 317 simple pattern 282 single quotation marks enclosing a construct 73, 96, 113, 131 skulker shell script 201 SMF (system management facilities) 222 socket address 179 sort shell command 206 sorting key example 207 source command 126 special characters 70, 93 parameters 116, 131 pattern 292 square brackets customization 305

square brackets (continued) wildcard expansion 75, 97 standard error **BPXBATCH 146** ddname 63. 87 file descriptor 63 ISPF shell 157 meaning 62, 86 redirection 64, 88 standard input **BPXBATCH 146** ddname 63, 87 file descriptor 63 ISPF shell 157 meaning 62, 86 redirection 64, 88 standard output **BPXBATCH 146** ddname 63, 87 file descriptor 63 ISPF shell 157 meaning 62, 86 redirection 63, 87 statement exit 296 if 296 next 296 return 123 status indicator location 13 meaning 20 job 137 print iob 249 STATUS TSO/E command 249 stderr file 146, 157 stdin file 63, 87, 146, 157 stdout file 63, 87, 146, 157 STEPLIB data sets 45, 58, 152 STEPLIB variable 45, 58 sterr file 63, 87 sticky bit symbolic and external links 183 STOP signal 139 storage not enough 25 stream closing 64 string manipulation function 297 value 284 variable, built-in 294 stty shell command 136 su shell command 80, 103 subcommand mode subcommands 22 using 22 subdirectory removing 222 submit shell command 145 SUBMIT TSO/E command 135, 249 submitting JCL 145 substitution command 69, 92

substring 111 summary of changes z/OS UNIX V2R2 xxii V2R3 xxii Summary of changes xxii superuser switching to 80, 103 whoami command 81, 104 symbolic link deleting 205 sticky bit 183 symbolic links command differences tar, du, find, pax, rm, ls 183 symbolic mode 219 syscall command 156 System Display and Search Facility 9 system management facilities 222 system-specific directories /etc, /tmp, /var, /dev 183

T

tab character awk 283 talk shell command 167 TCP/IP address for X-Window application 4 File Transfer Protocol (FTP) facility 27, 269 tcsh shell changing the locale 54 customizing 49 files accessed at termination 60 telnet from TSO/E 27 Temporary File System (TFS) 178 terminal 3270 11 ASCII interface 31 **EBCDIC** interface 11 terminal emulators 5 terminal file 179 test shell command 117 tilde (~) notation 192 time shell command 80, 102 time zone specifying 44, 57 Tivoli Storage Manager 211 tracing job 136 tracked alias 67 TRANSMIT TSO/E command 172 TSM file backup 211 tso command using 81 tso shell command in a shell script 108, 126 TSO/E address space working directory 185 case-sensitive processing 200 commands entering from ISPF 185
TSO/E (continued) commands (continued) entering from the shell 26 printing files 248 using a relative path name 185 ftp and telnet 27 invoking BPXBATCH 154 mail facilities 163 prefix 182 prompt 16 switching to 27 TSO/E command ALLOCATE 254 DELETE 270 HELP 185 ISPF 16 MKDIR 194 **OBROWSE 211** OCOPY 254, 259 **OGET 258** OGETX 261 OMVS 23 **OPUT 253 OPUTX 256** PRINTDS 249 **PROFILE PLANGUAGE 28 RECEIVE 172** SEND 172 STATUS 249 SUBMIT 135, 249 TRANSMIT 172 tsocmd command using 82 typeset shell command 111 TZ variable 44, 57

U

UTD 4294967294 223 changing 80, 103, 224 umask shell command 224 unalias shell command 68, 90 Unicode Services file tagging 190 porting 190 Universal Time Coordinated (UTC) 44, 57 UNIX-to-UNIX copy program (UUCP) 163 unnamed pipe 179 user catalog 179 classes 217 definition 217 user interface **ISPF 317** TSO/E 317 user-defined function 299 utility definition 3 UUCP commands code page conversion 169 daemons 168 file transfer from a remote site 171

UUCP (continued) file transfer (continued) to a remote site 169 to the local public directory 170 file transfer (multiple) to a remote site 170 file transfer status checking 171 files public directory 171 network, using 168 notification of file transfer 170 permissions 170 remote site running a command on 172 transferring a file to a 169 uucp shell command 169 uulog shell command 171 uuname shell command 169 uupick shell command 169, 171 uustat shell command $\overline{169}$, $\overline{171}$ uuto shell command 169 uux shell command 169, 172

V

value assigning to a variable 284 numeric 284 string 284 variable assigning value 284 associating attributes 111 built-in numeric 293 built-in string 294 environment BPX_SPAWN_SCRIPT 40 displaying <u>37, 51</u> ENV 38 LANG 41, 44, 55, 57 LC_ALL 41, 55 LC COLLATE 41, 55 LC CTYPE 41, 55 LC MESSAGES 41, 55 LC SYNTAX 43, 56 LIBPATH 40, 54 LOCPATH 44, 57 PATH 38, 52 TZ 44, 57 exporting allexport option 46 profile file 36 shell arithmetic calculation 109, 127 creating 108, 126 displaying definitions 112 variant characters 4, 43, 56, 305 vi editor adding text 230 advanced topics 275 arrow keys 229 backwards search 235 changing text 233 checking substitutions 279

vi editor (continued) combining files 277 command editing 78, 99 controlling indention 277 copying text 237 cursor moving 229, 231 cursor commands 232 default permissions 218 deleting text 233 determining line numbers 279 editing options setting up a command file 276 editing several files 276 editing source code 277 file recovered message 237 locating text 234 making substitutions 278 message file recovered 237 modes 229 moving text 236 pasting text 236 quitting a file 234 saving a file 234 searching backwards 235 for brackets 278 for strings 234 setting tab stops 275 setting up an options command file 276 special characters 235 specifying a range of lines 278 text adding 230 changing 233 copying 237 deleting 233 locating 234 moving 236 pasting 236 undoing a command 233 using abbreviations 275 vi/ex file recovered 237 viewing an archive 214 VSAM data set copying to a file 257

W

wait shell command <u>139</u> wall shell command <u>168</u> wc shell command <u>208</u> whence shell command <u>39</u> which shell command <u>53</u> while loop <u>121</u>, <u>133</u>, <u>296</u> whoami shell command <u>81</u>, <u>104</u> wildcard character preventing expansion <u>46</u>, <u>59</u> word count <u>208</u> working directory TSO/E address space <u>185</u> workstation, remote login <u>31</u> write shell command <u>166</u>

Х

X-Window TCP/IP workstation address <u>4</u> X-Window application running 4

Υ

yacc shell command locale modifications <u>41</u>, <u>54</u>

Ζ

z/OS Print Server lp command <u>248</u> z/OS UNIX summary of changes for V2R2 <u>xxii</u> summary of changes for V2R3 <u>xxii</u> z/OS XL C/C++ iconv utility <u>266</u>, <u>267</u>, <u>306</u> zero file 179



SA23-2279-30

