

Zentrale Variablenbelegung

Bitte auf dieser Seite alle zu belegenden Variablen doppelt anklicken und im Variablenmenü definieren; anschließend im Buchfenster mit Hilfe der Funktion „Formate von ...“ die einzelnen Kapitel des Buches zentral aus diesem Dokument versorgen.

Hinweise: Auf den Vorgabeseiten sollten keine Änderungen vorgenommen werden!

Die Seiten 1 bis 6 sind vom Fachredakteur auszufüllen, die Seite 7 ist für das Producing und kann ignoriert werden.

Hinweise zur Sprachendienst- oder Produktions-Übergabe finden Sie im Kapitel 6.1 des Redaktions-Handbuchs

Produktname: Reliant UNIX

Version: 5.44

Betriebssysteme: Reliant UNIX

Handbuchtitel: User's Guide

(mit Untertitel)

Handbuchtitel-

Ergänzung:

Handbuchtyp: User Guide

Der Langtitel, wie er sich aus den oberen sechs Variablen ergibt, sollte nicht mehr als 111 Zeichen in bis zu drei Zeilen umfassen.

Wenn er länger ist muß auf der Vorgabeseite für den Druckauftrag an UK33 der Handbuchtitel editiert und ein Kurztitel für den Eintrag in die Druckschriften-Neuerscheinungen angegeben werden.

Produktfamilie: UNIX

Produktfamilie_2.Zeile: keine

Soll die Produktfamilie auf der Titelseite einzeilig erscheinen, so definieren Sie nur die Variable *Produktfamilie*. Die Variable *Produktfamilie_2.Zeile* sollte in diesem Fall mit „**keine**“ belegt sein.

Soll die Produktfamilie auf der Titelseite zweizeilig erscheinen, so definieren Sie die Variable *Produktfamilie* und *Produktfamilie_2.Zeile*, z.B. belegen Sie bei der Produktfamilie *SINIXwindows* die Variable *Produktfamilie* mit *SINIX* und die Variable *Produktfamilie_2.Zeile* mit *windows*.

Ausgabestand: September 1997

Bestellnummer: U25903-J-Z145-1-7600

Produktnummer: 71028

zuletzt bearbeitet am: 12.09.1997

Bearbeitungsstand:

Fachredakteur: Ch. Krüger/ K. H. Peter

Labor: OEC ES EP 5

TelefonNr.: (089) 636-40477

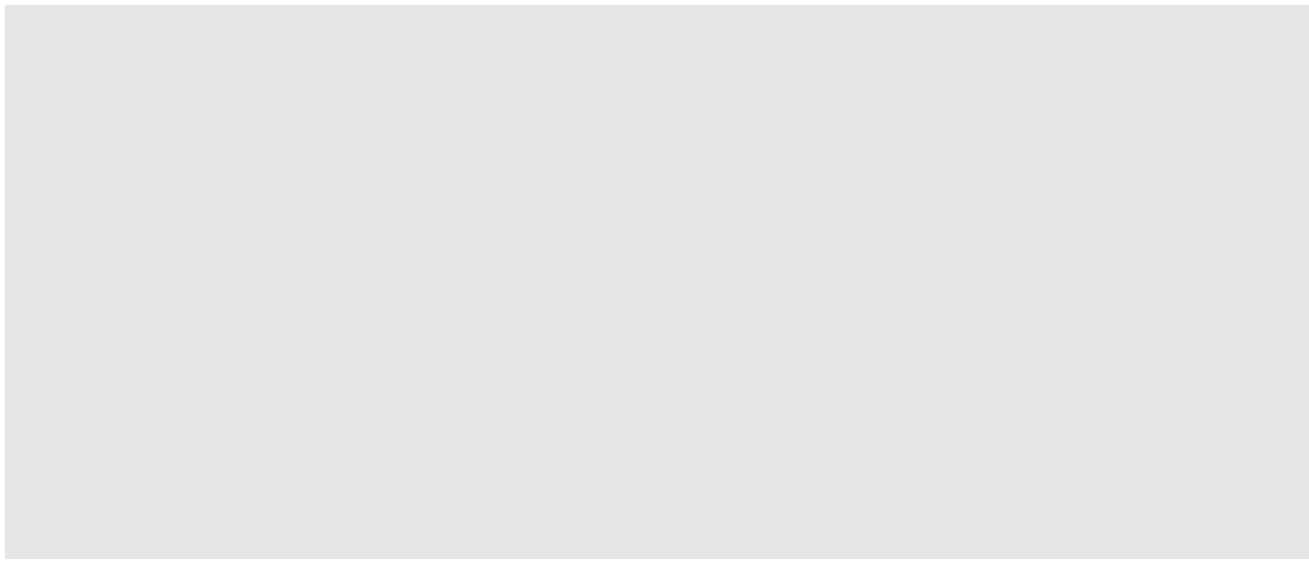
zuständiges

Producing-Labor OEC ES EP 5

SIEMENS
NIXDORF

Bestellnummer: U25903-J-Z145-1-7600

Produktnummer: 71028



Reliant UNIX 5.44

(Reliant UNIX)
User's Guide

User Guide

Zum Produkt gehören mehrere Handbücher

☐ nein

☐ ja, alle Titel für das Produkt liegen bei

Die von OEC ES EP 5 erstellten Titelseiten wurden ge-

Unterschriften: LL-FR _____

LL-OEC ES EP _____

Produktnummer: 71028

Bestellnummer: U25903-J-Z145-1-7600

Produkt: Reliant UNIX 5.44

Betriebssystem: Reliant UNIX

Handbuchtitel: User's Guide

Handbuchtyp: User Guide

Ausgabestand: September 1997

Redakteur: Ch. Krüger/ K. H. Peter

Labor: OEC ES EP 5

Zielgruppe (maximal 111 Zeichen, für Druckschriften-Neuerscheinung und Literaturpool):

SINIX beginners

Inhalt allgemein (maximal 259 Zeichen, für Druckschriften-Neuerscheinung und Literaturpool):

The SINIX User's Guide describes the most important elements of the SINIX operating system, including how to use SINIX, the file system, how processes work, and the shell.

Inhalt Speziell (maximal 222 Zeichen, für Druckschriften-Neuerscheinung):

The manual also describes the editors vi, ced, ed and sed, the awk programming language, network functions, electronic mail, archiving, off-line storage devices, and the basics of Spool.

Von OEC ES EP 5 auszu-

Eintrag in

Druckschriften-Neuerscheinung:

Literaturpool:

Sprachendienstübergabe:

Lieferung an den Sprachendienst über OEC ES EP 5

Bestellnummer: U25903-J-Z145-1-7600

Produktnummer: 71028

Handbuchtitel: Reliant UNIX 5.44, User's Guide

Fachredakteur: Ch. Krüger/ K. H. Peter

Labor: OEC ES EP 5

Es handelt sich bei dieser Übergabe um eine

☐ Erstlieferung

Folgende Formulare müssen bei Erstlieferung übergeben werden:

☒ In deutscher Sprache ausgefülltes Formular „Druckschriften-Neuerscheinung/Literaturpool“ . (siehe Master-Datei)

☒ In deutscher Sprache ausgefülltes Titelei-Formular. (siehe Master-Datei)
Gibt es mehrere Handbücher zu einem Produkt, werden bei der ersten Sprachendienst-Übergabe des ersten Handbuchs die Titelei-Formulare für alle folgenden Handbücher mit abgegeben, um eine einheitliche Titelei zu gewährleisten.

☐ Nachlieferung

Anzahl der zu übersetzenden Seiten

Folgende Dateien sind an den Sprachendienst zu übergeben:

FrameMaker

☐ **Manuskriptdatei** (deutsch oder Mischdatei)
Mit der Prozedur **fmueber** auf die Übergabekennung Sprachendienst auf dem Service-Rechner **qm2drive** übertragene Datei.
Standardname: <Produkt-Nr.>.TAR.Z

☐ **Konvertierte engl. Frame-Datei**
Mit der Prozedur **fmueber** auf die Übergabekennung Sprachendienst auf dem Service-Rechner **qm2drive** übertragene Datei.
Standardname: Z.<Produkt-Nr.>.TAR.Z

☐ **Vorgängerdatei** (englisch):
Produktnummer des engl.
Vorgängerhandbuchs:

Bestellnummer des engl.
Vorgängerhandbuchs:

ICP

☐ **Manuskriptdatei** (deutsch oder Mischdatei)
Mit der Prozedur **fmueber** auf die Übergabekennung Sprachendienst auf dem Service-Rechner **qm2drive** übertragene Datei.
Standardname: <Produkt-Nr.>.TAR.Z

☐ **Vorgängerdatei** (englisch):
Produktnummer des engl.
Vorgängerhandbuchs:

Bestellnummer des engl.
Vorgängerhandbuchs:

Datum:

Unterschrift Fachredakteur:

QM2-Grafik-Auftrag (intern)

vom FR auszufüllen!

Handbuch- titel:	User's Guide				
Produkt-Nr:	71028	Bestell-Nr:	U25903-J-Z145-1-7600		
Redakteur:	Ch. Krüger/ K. H.	Dienststelle:	OEC ES EP 5	Tel.:	(089) 636-40477
Übersetzer:					
Satz- beginn:		Prod.- beginn:		SNI- Format	
Anzahl Grafiken:		neu		ändern	Termin:

Grafiken erstellt von: _____

Grafiken erstellt mit: ☐ Designer ☐ FrameMaker ☐

Dateiverzeichnis: _____

Datei gesichert auf: ☐ Festplatte ☐ Diskette ☐ Bernoulli ☐

Datei gesichert als: ☐ .drw ☐ .doc ☐ .txt ☐ .eps ☐

Datei(en) archiviert: ☐ qm2tulpe ☐ BS2000 ☐

Grafiken an FR:

Grafiken an SD: ☐ übersetzen ☐ Korrektur lesen

Bemerkungen:

vom FR auszufüllen!

Datum/Unterschrift

Auftrag erledigt:

Datum/Unterschrift

Druckauftrag

SIEMENS
NIXDORF

Produktnummer: 71028 Bestellnummer: U25903-J-Z145-1-7600

Handbuchtitel: Reliant UNIX 5.44 (Reliant UNIX)
User's Guide
User Guide

Betriebssysteme: Reliant UNIX

Labor: OEC ES EP 5

Redakteur: Ch. Krüger/ K. H. Peter

Telefon: (089) 636-40477

Die Druckvorlagen wurden erstellt mit dem Textsystem:

☐ Frame 4.0 ☐ Frame 3.0 ☐ ICY ☐ _____

Auflagenhöhe:

Vom Producing auszufüllen

Seitenanzahl (gesamt):

Fabrikategruppe:

Liefertermin:

B70:

Bemerkungen:

Datum:

Unterschrift:

Produktnummer: 71028

Bestellnummer: U25903-J-Z145-1-7600

	j	n
X/Open-Logo		
Validierung		
Warenzeichen		

Aufwand	j	n
Bildmontagen		
Raster		
2. Farbe		

Außenformat:

- ☐ 19 x 24 cm ☐ 12 x 21 ☐ _____
- ☐ DIN A4 ☐ DIN A5

Druckvorlagen Handbuch:

- ☐ PS-Datei
- ☐ Papier
- ☐ _____

Druckvorlagen Titelei:

- ☐ PS-Datei
- ☐ Papier
- ☐ _____

Angaben zur PostScript-Datei (Handbuch)

Dateiname

Datenträger (Bitte zurück!)

- ☐ Magnetbandkassette (UNIX)
- ☐ 3.5 Diskette (MS-DOS)

Textverarbeitungssystem

- ☐ FrameMaster V3.0 auf UNIX
- ☐ FrameMaster V3.0 auf MS-DOS
Druckertreiber: _____
- ☐ FrameMaster V4.0 auf UNIX
- ☐ FrameMaster V4.0 auf MS-DOS
Druckertreiber: _____
- ☐ _____

Zeichensätze in der PostScript-Datei enthalten

Datei komprimiert

- ☐ nein
- ☐ ja
- ☐ selbst-dekomprimierend
- ☐ dekomprimieren mit: _____
- ☐ _____

Angaben zur PostScript-Datei (Titelei)

Dateiname

Datenträger (Bitte zurück!)

- ☐ Magnetbandkassette (UNIX)
- ☐ 3.5 Diskette (MS-DOS)

Textverarbeitungssystem

- ☐ FrameMaster V3.0 auf UNIX
- ☐ FrameMaster V3.0 auf MS-DOS
Druckertreiber: _____
- ☐ FrameMaster V4.0 auf UNIX
- ☐ FrameMaster V4.0 auf MS-DOS
Druckertreiber: _____
- ☐ _____

Zeichensätze in der PostScript-Datei enthalten

Datei komprimiert

- ☐ nein
- ☐ ja
- ☐ selbst-dekomprimierend
- ☐ dekomprimieren mit: _____
- ☐ _____

Reliant UNIX 5.44

User's Guide

Edition September 1997

Copyright and Trademarks

SINIX® Copyright © Siemens Nixdorf Informationssysteme AG 1990.

SINIX is the UNIX® System derivative of Siemens Nixdorf Informationssysteme AG.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

Copyright © Siemens Nixdorf Informationssysteme AG 1995.

All rights reserved.

Delivery subject to availability; right of technical modifications reserved.

All hardware and software names used are trademarks of their respective manufacturers.

Introduction

Basics for SINIX system users

The file system

Processes

The shell

The display editor vi

Other SINIX editors

The awk programming language

Network user functions

Electronic mail

Continued ►

Archiving

Off-line storage devices

Printing with Spool

Appendix

Reference section and index

Contents

Volume 1:

1	Introduction	1
1.1	Target group	1
1.2	Summary of contents	2
1.3	Requirements	4
1.4	Layout	5
2	Basics for Reliant UNIX	
	system users	9
2.1	Basics	9
2.1.1	Structure of Reliant UNIX	9
2.1.2	Main characteristics of Reliant UNIX	10
2.1.3	Reliant UNIX concepts	10
2.1.4	The Native Language System NLS	17
2.2	Logging in	17
2.2.1	Terminal	18
2.2.2	Login name	18
2.2.3	Password	18
2.2.4	Problems during login	20
2.3	Communicating with the system	21
2.3.1	The shell	21
2.3.2	Keyboard input	21
2.4	Logging out	25
2.5	Summary	26
3	The file system	27
3.1	Introduction	27
3.2	The structure of the file system	27
3.2.1	Files	28
3.2.2	Directories	30
3.2.3	The hierarchical structure of the file system	30
3.2.4	The login directory	32
3.2.5	The current directory	34
3.2.6	Path names	35
3.2.7	Moving to another directory: the cd command	42
3.2.8	Permissions for files and directories	43
3.2.9	File types in Reliant UNIX	51

Contents

3.3	Organizing a directory system	51
3.3.1	Creating a directory: the mkdir command	52
3.3.2	Listing the contents of a directory: the ls command	53
3.3.3	Creating files using the cat and echo commands	58
3.3.4	Deleting a file: the rm command	61
3.3.5	Deleting a directory: the rmdir command	62
3.4	Accessing files	64
3.4.1	Displaying the contents of a file: the cat, pg and pr commands	64
3.4.2	Copying a file: the cp command	73
3.4.3	Moving and renaming files: the mv command	75
3.4.4	Counting lines, words and characters in a file: the wc command	78
3.4.5	Changing the owner of a file: the chown command	80
3.4.6	Comparing files: the diff command	81
3.4.7	Searching for a pattern in a file: the grep command	84
3.4.8	Sorting and merging files: the sort command	86
3.5	Summary	89
4	Processes	91
4.1	Introduction	91
4.2	Processes and process commands	92
4.2.1	The process ID (PID)	92
4.2.2	Real and effective user ID and group ID of a process	92
4.2.3	Obtaining information on processes	93
4.2.4	Process priority	95
4.2.5	Suspending processes	95
4.2.6	Aborting active processes	96
4.2.7	Background processes	97
4.2.8	The nohup command	98
4.3	Summary of commands for process control	99
5	The shell	101
5.1	Introduction	101
5.2	The shell command language	101
5.2.1	Entering command lists	102
5.2.2	Redirecting input and output	104
5.2.3	Filters and pipes	107
5.2.4	Shell metacharacters	110
5.3	Shell programming	117
5.3.1	Shell scripts	117
5.3.2	Variables	120
5.3.3	Assigning a value to a variable	126
5.3.4	Shell programming constructs	130
5.3.5	Debugging shell scripts	147

5.4	Modifying your login environment	149
5.4.1	The .profile file	149
5.5	Exercises	154
5.5.1	Command language exercises	154
5.5.2	Shell programming exercises	155
5.6	Solutions	156
5.6.1	Solutions to the command language exercises	156
5.6.2	Solutions to the shell programming exercises	157
5.7	Summary of the shell command language	160
5.8	Summary of shell programming constructs	163
6	The display editor vi	167
6.1	Introduction	167
6.1.1	What vi does	167
6.2	Getting started	169
6.2.1	Setting your login environment	169
6.3	Creating a file	172
6.3.1	Operating modes	173
6.3.2	Creating text in input mode	174
6.4	Creating text	175
6.4.1	Appending text	175
6.4.2	Inserting text	176
6.4.3	Inserting a blank line	178
6.5	Editing text in command mode	181
6.5.1	Positioning the cursor	182
6.5.2	Deleting text	185
6.6	Quitting vi	187
6.6.1	Summary	189
6.7	Positioning the cursor	189
6.7.1	Positioning the cursor on a character	190
6.7.2	Line-oriented positioning	195
6.7.3	Word-oriented positioning	196
6.7.4	Sentence-oriented positioning	201
6.7.5	Paragraph-oriented positioning	203
6.7.6	Screen-oriented positioning	205
6.8	Positioning the cursor in undisplayed text	208
6.8.1	Scrolling text	209
6.8.2	Moving the cursor to a specific line	211
6.8.3	Line numbers	211
6.8.4	Searching for patterns (the / and ? commands)	212
6.9	Deleting text	216
6.9.1	Deleting text in input mode	216
6.9.2	Undoing the last command	218
6.9.3	Deleting text in command mode	218

Contents

- 6.10 Modifying text 223
 - 6.10.1 Replacing text (the r command) 223
 - 6.10.2 Substituting text (s command) 224
 - 6.10.3 Changing text (c command) 226
- 6.11 Cutting and pasting text 228
 - 6.11.1 Moving text 229
 - 6.11.2 Correcting transposed characters 229
 - 6.11.3 Copying text 230
 - 6.11.4 Copying or moving text using named buffers 231
- 6.12 Special commands 232
 - 6.12.1 Repeating the last command 232
 - 6.12.2 Joining lines together 233
 - 6.12.3 Clearing and redrawing the screen 233
 - 6.12.4 Changing lowercase to uppercase and vice versa 234
- 6.13 Line editor commands 234
 - 6.13.1 Returning temporarily to the shell 234
 - 6.13.2 Writing text to a file 235
 - 6.13.3 Positioning to a specific line 236
 - 6.13.4 Deleting the remaining lines in the buffer 236
 - 6.13.5 Reading a file into the buffer 236
 - 6.13.6 Global changes 237
- 6.14 Quitting vi 239
- 6.15 Special options 240
 - 6.15.1 Restoring a file after a system crash 240
 - 6.15.2 Editing more than one file 241
 - 6.15.3 Viewing file contents 241
- 6.16 Exercises 243
 - 6.16.1 Exercise 1 243
 - 6.16.2 Exercise 2 244
 - 6.16.3 Exercise 3 245
 - 6.16.4 Exercise 4 246
 - 6.16.5 Exercise 5 246
 - 6.16.6 Exercise 6 247
- 6.17 Solutions 247
 - 6.17.1 Exercise 1 247
 - 6.17.2 Exercise 2 249
 - 6.17.3 Exercise 3 250
 - 6.17.4 Exercise 4 251
 - 6.17.5 Exercise 5 252
 - 6.17.6 Exercise 6 253
- 6.18 vi command summary 254

7	Other Reliant UNIX editors	259
7.1	ced	260
7.1.1	Creating text	260
7.1.2	Saving text	262
7.1.3	Moving the cursor	262
7.1.4	Deleting and inserting characters	263
7.1.5	Deleting and inserting lines	263
7.1.6	Quitting ced	264
7.2	ed	264
7.2.1	Creating text in input mode	264
7.2.2	Editing text in command mode	266
7.2.3	Saving text	268
7.2.4	Quitting ed	269
7.2.5	Using ed scripts	269
7.3	sed	270
7.3.1	Specifying sed commands	271
7.3.2	Editing text	272
7.3.3	Saving text	275
7.3.4	Editing text in a pipeline	277
7.4	Summary of other Reliant UNIX editors	278

Volume 2:

8	The awk programming language	283
8.1	Introduction	283
8.2	awk program structure	284
8.3	Calling awk	285
8.4	Patterns	286
8.4.1	The first type of pattern: relational expressions	287
8.4.2	The second type of pattern: regular expressions	289
8.4.3	The third type of pattern: BEGIN and END	292
8.4.4	Compound patterns	293
8.4.5	Pattern ranges	294
8.5	Actions	294
8.5.1	Processing the input file with simple actions	295
8.5.2	Processing the input file with complex actions	296
8.6	Input	312
8.6.1	Files and pipes	312
8.6.2	Input separators	313
8.6.3	Multiline records	314
8.6.4	The getline function	314
8.6.5	Command-line arguments	317

Contents

8.7	Output	318
8.7.1	Simple output using the print function	318
8.7.2	Formatted output using the printf function	319
8.7.3	Output separators	321
8.7.4	Redirecting output to files	322
8.7.5	Piping output to other commands	322
8.8	Using awk with other commands and the shell	323
8.8.1	The system function	324
8.8.2	awk and the shell	324
8.9	Sample applications	326
8.9.1	Some useful one-liners	326
8.9.2	Error messages	327
8.9.3	More sample applications	327
8.10	awk summary	332
9	Network user functions	339
9.1	Using the Sifmllan user interface	340
9.2	Logging in to a remote host	341
9.2.1	Logging in to a remote host via Sifmllan	341
9.2.2	Logging in to a remote host with rlogin	342
9.2.3	Terminating a session on the remote host	343
9.2.4	Interrupting an rlogin connection	344
9.2.5	Querying a password	345
9.2.6	Logging in to a remote host with telnet	345
9.2.7	Interrupting a telnet connection	346
9.3	Copying files between hosts	347
9.3.1	Copying files via Sifmllan	348
9.3.1.1	Fetching files from a remote host	348
9.3.1.2	Sending files to a remote host	350
9.3.2	File transfer with ftp	351
9.3.2.1	Displaying a list of the files on the remote host	352
9.3.2.2	Copying files with get and put	352
9.3.2.3	Copying multiple files with mget and mput	353
9.3.2.4	Quitting an ftp session	353
9.3.2.5	Aborting ftp during file transfer	354
9.3.3	Copying files between hosts with rcp	354
9.3.3.1	Copying a remote file to the local host with rcp	354
9.3.3.2	Copying a local file to a remote host with rcp	355
9.3.3.3	Copying directories with rcp	356
9.3.4	Executing commands on remote hosts	356
9.3.4.1	Using shell metacharacters	357
9.3.4.2	Calling rsh without a command argument	358
9.3.4.3	Calling rsh under a different name	358

9.4	Obtaining network information	359
9.4.1	Listing defined hosts in the network	359
9.4.2	Displaying the active sessions in the network	359
9.4.3	Displaying the status of the hosts in the network	360
9.4.4	Obtaining information on the Network Information Service	361
9.4.5	Displaying user information with finger	362
9.4.6	Querying the operating status of a host with ping	363
9.5	What do I do if	365
9.5.1	A host cannot be accessed	365
9.5.2	Access to a remote host is denied	366
10	Electronic mail	367
10.1	Introduction	367
10.2	The mail command	367
10.2.1	Overview of the mail command	367
10.2.2	Sending messages using mail	368
10.2.3	Receiving messages with mail	371
10.2.4	Summary of the mail command options	375
10.3	The mailx command	375
10.3.1	Summary of the mailx command	375
10.3.2	Sending messages with mailx	376
10.3.3	Receiving messages with mailx	383
10.3.4	Summary of the mailx command options	391
10.4	The .mailrc file	392
11	Archiving	397
11.1	Introduction	397
11.2	Archiving files and directories	397
11.2.1	Archiving using the tar command	398
11.2.2	Archiving using the cpio command	401
11.3	Listing archive contents	403
11.3.1	Listing archive contents using tar	403
11.3.2	Listing archive contents using cpio	405
11.4	Extracting files and directories from an archive	406
11.4.1	Extracting files and directories using tar	406
11.4.2	Extracting files and directories using cpio	407
11.5	Conversion	409
11.5.1	Converting files using the dd command	409
11.5.2	Copying files between Reliant UNIX and DOS using doscp	411
11.6	Summary of the archiving and conversion commands	414

Contents

12	Off-line storage devices	417
12.1	Introduction	417
12.2	Types of off-line storage device	417
12.2.1	Floppy disk drives	418
12.2.2	Tape cartridge drives	419
12.2.3	CD-ROM drives	419
12.2.4	WORM drives	420
12.2.5	Hard disk drives	420
12.2.6	Magnetic tape drives	420
12.3	Using floppy disks	421
12.3.1	Handling floppy disks	421
12.3.2	Distinguishing features of floppy disks	421
12.3.3	Write protection	424
12.3.4	Using a floppy disk drive	425
12.3.5	Formatting a floppy disk using the flformat command	425
12.3.6	Formatting a floppy disk using the dosformat command	426
12.4	Using tape cartridges	428
12.4.1	Handling tape cartridges	428
12.4.2	Write protection	428
12.4.3	Using a tape cartridge drive	429
12.4.4	Processing tape cartridges using the mt command	429
12.5	Devices and special files	432
12.5.1	Displaying SCSI devices: the gettypes command	432
13	Printing with Spool	435
13.1	Spool from the user's point of view	435
13.2	Spool elements	438
13.2.1	Spool objects	438
13.2.2	Options for printing documents	441
14	Appendix	445
14.1	Basic Reliant UNIX commands	445
14.2	Regular expressions	449
14.3	ISO 646 character set	458
Glossary		465
List of figures		485
List of tables		487
Related publications		493
Index		503

1 Introduction

Up to now, the “Reliant UNIX” operating system was called “SINIX”. In Version 5.43 this name has been changed to “Reliant UNIX” as a result of the UNIX versions from Siemens Nixdorf and Pyramid Technology being amalgamated. The name “Reliant UNIX” reflects the stringent demands on reliability and availability which our standardized software meets, both in commercial and technical applications.

The operating system is the link between you and your computer. It controls the complex components of a computer and provides you with a user-friendly interface to them.

Reliant UNIX is characterized by the following features:

- Reliant UNIX is an interactive operating system, i.e. the user and the computer communicate directly with one another.
- Reliant UNIX is a multiuser operating system, i.e. it allows several users to work with the same computer at the same time.
- Reliant UNIX has multitasking capability, i.e. it can process several jobs at the same time.

1.1 Target group

This manual provides introductory information on using Reliant UNIX. It is therefore aimed primarily at newcomers to Reliant UNIX who have little or no experience of this operating system. Users who already have a general knowledge of computing will find it easier to understand the topics covered in this manual.

The second part of the manual provides this target group with information that is fundamental to using Reliant UNIX, whereas the third part introduces more specialized topics.

This *User's Guide* also serves as a comprehensive reference work for more experienced Reliant UNIX users.

References are made in the individual chapters to other relevant publications. In addition, the appendix (part 4) contains a list of references.

1.2 Summary of contents

This manual is divided into four parts and contains thirteen chapters, an appendix and a reference section and index.

Part 1 Introduction	
Chapter 1:	Introduction
Part 2 General information on Reliant UNIX	
Chapter 2:	Basics for Reliant UNIX system users
Chapter 3:	The file system
Chapter 4:	Processes
Chapter 5:	The shell
Part 3 Specific aspects of Reliant UNIX	
Chapter 6:	The display editor vi
Chapter 7:	Other Reliant UNIX editors
Chapter 8:	The awk programming language
Chapter 9:	Network user functions
Chapter 10:	Electronic mail
Chapter 11:	Archiving
Chapter 12:	Off-line storage devices
Chapter 13:	Printing with the Spool of Reliant UNIX
Part 4 Appendix, reference section and index	
Appendix	Basic Reliant UNIX commands Regular expressions ISO 646 character set
Reference section and index	Glossary List of figures List of tables Related publications Index

Part 1

Part 1 contains chapter 1, which summarizes the target group, the structure and the contents of this guide and describes its layout.

Part 2

Part 2 comprises chapters 2 through 5 and describes the main features of the operating system. The descriptions in part 3 of this guide build on what you have learned in part 2.

- In chapter 2, *Basics for Reliant UNIX system users*, you will become familiar with the basic features of Reliant UNIX and learn how to log in.
- Chapter 3, *The file system*, provides introductory information on using the file system. It describes important commands for building your own directory hierarchy and accessing files.
- Chapter 4, *Processes*, explains the concept of the Reliant UNIX process.
- Chapter 5, *The shell*, describes the function of the shell both as a command interpreter and as a programming language.

Part 3

Part 3 comprises chapters 6 through 12 and introduces specific topics.

- Chapter 6, *The display editor vi*, shows you how to create text using the *vi* editor.
- Chapter 7, *Other Reliant UNIX editors*, provides information on the Reliant UNIX editors *ced*, *ed* and *sed*.
- Chapter 8, *The awk programming language*, introduces you to the syntax, lexical units and expressions of the *awk* programming language.
- Chapter 9, *Network user functions*, describes the necessary commands for working with Reliant UNIX in networks.
- Chapter 10, *Electronic mail*, introduces you to the commands that enable you to exchange messages under Reliant UNIX.

Introduction

- Chapter 11, *Archiving*, describes the most frequently used Reliant UNIX commands for archiving and extracting your data.
- Chapter 12, *Off-line storage devices*, gives you information on using storage media and drives.
- Chapter 13, *Printing with Reliant UNIX Spool*, shows you how to print out your data under Reliant UNIX.

Part 4

Part 4 - the appendix, reference section and index - contains the following sections:

- Basic Reliant UNIX commands
- Regular expressions
- ISO 646 character set
- Glossary
- List of figures
- List of tables
- Related publications
- Index

1.3 Requirements

The performance of the Reliant UNIX operating system is governed by various factors. To ensure that both your computer and the operating system run as smoothly as possible, the following requirements must be met:

- Some of the services that Reliant UNIX offers must first be set up by the system administrator and thereafter require continual maintenance. One example of this is the *mail* system, which allows you to send and receive mail across both local and remote networks.

- Some commands are restricted to specific hardware environments. These commands will run only on specific types of system or terminal, or are associated with optional hardware components. For example, you can only use the functions described in chapter 13 *Printing with Reliant UNIX Spool* if your system is equipped with a printer. On the other hand, your system may provide functions that are not included in the basic Reliant UNIX package. One example of this is the full-screen editor *MAXed*, which provides a wide range of text processing functions.

1.4 Layout

The following layout conventions are used throughout this guide:

- Inner running title: The inner running title gives the name of the chapter.
- Outer running title: The outer running title gives the name of the section.

User input

- All examples of entries made by the user require the  key to be pressed at the end of each line. For this reason,  is not printed.
- Within descriptive text, user input is printed in *italics*.
- Function keys are represented by the relevant key symbol. For example, the key symbol  represents the control key, and the key symbols  and  represent the arrow keys you use to move the cursor. Where combinations of function keys and alphanumeric characters occur, e.g.  d, the function keys are represented by symbols, and the alphanumeric characters appear in a *fixed-pitch font*. Remember that Reliant UNIX differentiates between uppercase and lowercase in your input.

Output

- Output from the operating system is printed in a *fixed-pitch font* unless it appears within descriptive text, in which case it is printed in *italics*.

Command description

In chapters that describe commands in some detail, the layout is as follows:

- The standard syntax of a command appears in a `fixed-pitch font`. The variables of a command line appear in *italics*. You must replace each variable with an appropriate value (e.g. a file name). Characters that are not printed in italics, however, must be entered exactly as shown.
- Within descriptive text, command input (including the command options and arguments), command output, variables, file names and path names are all printed in *italics*. No distinction is made in this case between constants and variables.
- Options: options modify the way in which a command functions. They generally consist of a letter preceded by a hyphen (-).

The following rules apply:

– You can list options in any order.

– There are two ways of specifying options without arguments:

singly: `command -a -b -c`

grouped: `command -abc`

– Options that require an argument, e.g. *-dargument* or *-fargument*, cannot be combined with those that do not. Options with arguments must be entered individually, separated by blanks:

`command -abc -dargument -f argument`

Any deviations from these rules are indicated in the relevant command description.

- Optional input, i.e. input that is not essential for command execution, is enclosed in square brackets []. Do not enter the square brackets.
- Ellipses ... indicate that the preceding expression can be repeated.
- The dollar symbol (\$) is used to represent the shell prompt in the command examples.

This guide does not attempt to cover the full functionality of each command. If you require more detailed information on commands, refer to the relevant entries in the *Commands* manual [5].

Command summary

The explanation of each command ends with a tabular summary of the functions, arguments and options of the command covered in the descriptive text.

The full syntax of the Reliant UNIX command involved is given at the end of each command summary. This syntax line contains all the arguments and options of a command and has the following format:

- The command name and the characters you must enter exactly as shown are printed in **bold**.
- Variables are printed in *italics*. You must replace each variable with an appropriate value (e.g. a file name).
- Optional input, i.e. input that is not essential for command execution, is enclosed in square brackets []. Do not type in the square brackets themselves.
- Ellipses ... indicate that the previous expression can be repeated.
- A mandatory blank is represented by the character _.
- Options: the syntax line contains all possible options of a command.

The following rules apply:

- You can list options in any order.
- There are two ways of specifying options without an argument:

singly: `command -a -b -c`

grouped: `command -abc`

- Options that require an argument, e.g. *-dargument* or *-f argument*, cannot be combined with those that do not. Options with arguments must be entered individually, separated by blanks:

`command -abc -dargument -f argument`

- If several options are enclosed in braces {}, you must specify one of these options.
- The or character | between two options means that you must specify one or the other of these options.

Introduction

Cross-references

The following examples illustrate the forms in which reference is made to other chapters or sections in this guide or in other manuals, and to other relevant publications.

Other chapters or sections	
see chapter 2 <i>The file system</i>	Reference to chapter 2, entitled <i>The file system</i> , in this guide.
see section 3.2.5 <i>The current directory</i>	Reference to section 3.2.5 <i>The current directory</i> in this guide
Other Reliant UNIX manuals	
see the <i>System Administrator's Guide</i> [6]	Reference to the manual listed as number [6] at the end of this guide.

Examples

Examples are not identified in any special way. Note that the data used in the examples is not a part of your system. If you want to try the examples yourself, you must first create the relevant files, directories, and so on. You are given detailed instructions where appropriate in the text.

Notes and cautions

Note: is used to draw your attention to important information.

Caution: is used to warn you of potential data loss or damage to devices.

2 Basics for Reliant UNIX system users

2.1 Basics

In order to use Reliant UNIX effectively, it is important for you to understand its basic structure, its main characteristics and above all its underlying concepts. This section provides you with a brief insight into these three areas. It also introduces you to the *Native Language System* (abbreviated to *NLS*), which makes it possible to internationalize and localize Reliant UNIX.

You will find more detailed information on the concepts, together with many illustrative examples, in later chapters of this guide. Reference is made to the relevant chapters in the description of each concept.

2.1.1 Structure of Reliant UNIX

Reliant UNIX is an operating system. The purpose of an operating system is to provide the user of a computer with access to the machine itself via an interface that is easy to understand and use, and to administer the communication between user and machine. The Reliant UNIX operating system consists of

- the kernel
- the user's working environment

The *kernel* controls centralized mechanisms, e.g. processors, terminals and printers, and allocates CPU time to processes. You access the functions of the kernel by means of system calls. There are two ways of using system calls. You can either specify them directly in a program you have written yourself or access them indirectly via the individual components of the working environment supplied by the system.

The *working environment* is made up of system programs, application programs and user programs. System programs are all the commands that are supplied as a standard part of the system software. Application programs, on the other hand, are software products such as word processors that are not supplied as standard and must be installed separately. Finally, user programs are tools written by users themselves, for example to calculate wages or display appointments on the screen at regular intervals.

A particularly important aspect is that the system programs are not privileged in comparison with user programs. This means that you can write your own version of any system program using the resources provided by the working environment.

2.1.2 Main characteristics of Reliant UNIX

Reliant UNIX is characterized by the following important features:

- It is *interactive*.
- It has *multitasking* capability.
- It is a *multiuser* system.

An *interactive* system is a system that conducts a dialog with the user. You can enter any amount of data and commands, according to your requirements, and thus determine the type, sequence and frequency of your own input. The system responds either immediately or once you have made further necessary entries.

An operating system with *multitasking* capability can process more than one job at the same time.

A *multiuser* system enables a single computer to be shared by several users.

2.1.3 Reliant UNIX concepts

Reliant UNIX is based on a number of concepts that enable users to customize their application and development environment. Specifically, these concepts are:

- the file concept
- the process concept
- the command interpreter
- redirection of input and output
- filters, pipes and pipelines
- regular expressions

The file concept

Reliant UNIX has a file system with a hierarchical structure. A file can consist of between zero and almost any number of bytes. In Reliant UNIX, all software and hardware components that are accessed by the user are administered as files. In this context, software components are understood to be kernel components, system and user programs, other data stored by the user and control data generated during operation. Hardware components, on the other hand, are all the physical components of a computer, such as terminals, printers and floppy disk drives, that users can access by means of software components.

By working with one or more of the components, users cause the corresponding files to be processed (opened, read, written, closed).

From the point of view of the user, these files are distinguished first and foremost by their position in the hierarchy of the file system and by their name. From the point of view of the system, they are distinguished by their file type, which is assigned automatically. The file type tells the operating system whether the file contains data or is a special file. If it is a special file, the operating system performs the relevant operation on the device, e.g. writes to a floppy disk in the floppy disk drive.

One of the main advantages of this homogenous file concept is that all the commands for processing files are also applicable to devices.

More information on files, file types and the file system is given in chapter 3 *The file system*. You will find details on devices and special files in chapter 12 *External storage devices*.

The process concept

A process is a program that is at some stage of execution. A program is understood to be a set of related data and commands. Each program is stored in a file. On execution of such a file, a process is created. All active programs are therefore processes that are in turn capable of creating further new processes.

All processes belong to a hierarchical system with a structure similar to that of the file system described above. Processes form a hierarchy because every process has a creator, known as the *parent process*. Each process can in turn spawn further processes, known as *child processes*.

Once you have logged in to Reliant UNIX, a process runs automatically on your terminal. This process acts as a command interpreter, i.e. it reads the commands you type in and executes the associated program as a child process. The command interpreter normally waits until the child process has

terminated before processing new input from the user. This type of child process is known as a *foreground process*. If, however, you want a number of processes to run at the same time, you can tell the command interpreter explicitly not to wait until termination of the child process. This type of child process is known as a *background process*.

For further information on processes, see chapter 4 *Processes*. Section 2.2 *Logging in* in this chapter tells you how to log in to your terminal.

Command interpreter

Reliant UNIX provides command interpreters as standard user interfaces. Command interpreters are not part of the kernel but belong to the working environment, i.e. they are system programs that can be replaced if necessary. The function of a command interpreter is to read, interpret and execute the commands entered by the user. It therefore acts as a direct link between the user and the operating system or machine.

A default command interpreter is set for each user. It starts automatically as the first process when you log in at your terminal. This process continues to run until you terminate it explicitly, i.e. until you log out. If you require the functions of a different command interpreter, you can have your system administrator change the default permanently. Alternatively, you can call the other command interpreter just as you would call any other command. This starts a new process which also runs until you terminate it with an appropriate command. You then return to your default interpreter.

The most important command interpreters in Reliant UNIX are:

- the Bourne shell *sh*
- the Korn shell *ksh*
- the C shell *csh*

These shells differ in terms of their functionality and this is reflected in the relative ease with which they can be learned. The simplest shell is the Bourne shell.

Each of these shells has a number of special features. For example, they allow you to execute a command without starting a new process or to concatenate commands in order to perform complex operations in a single step.

These special features are:

- built-in commands
- the ability to recognize and process a wide variety of special characters

Built-in commands

Built-in commands are used in the same way as normal commands. The fundamental difference between these two types of command is that shell built-in commands are subprograms and form part of the shell program itself. When you call one of these commands, the shell executes the corresponding subprogram itself without creating a new process for it. Built-in commands therefore execute faster than commands of the same name which are not built-in. For this reason, the shell executes its own built-in commands in preference to normal commands of the same name.

These commands are known as *shell built-ins* or *special commands*.

The Bourne shell built-ins are listed under *sh* in the *Commands* manual [5]. They are described in alphabetical order together with the other commands. The built-in commands of the other shells are also described in the *Commands* manual [5] under the headings *ksh* and *csh* respectively.

Metacharacters

Each shell can process metacharacters from the command line. These metacharacters can be divided into different functional groups, the most important of which are:

- Metacharacters used to separate the individual arguments in the command line to enable the shell to recognize them as such. The most important and most frequently used metacharacter in this group is the blank.
- Metacharacters used to abbreviate arguments or to specify argument patterns, similar to regular expressions. For example, you can use these metacharacters to apply a command to a number of files whose names match a pattern. These metacharacters include the asterisk *, which stands for any number of characters, the question mark ?, which stands for exactly one character, and the square brackets [], which stand for exactly one character from the set specified in the brackets.

- Metacharacters used to replace a command specified as an argument with the output of this command. Backquotes ‘ ‘ are used for this purpose. If you specify a command as an argument of another command in these metacharacters, the argument command is first executed and replaced with its result. The actual command is then executed with this result as an argument.
- Metacharacters used to partially or completely escape (i.e. override) the special meaning of a character. You require these characters if, for example, you have to specify file names that contain one or more metacharacters. The backslash \ enables you to escape the next special character only. You use single quotes ‘ ’ to escape all the characters enclosed in them. Double quotes “ ” escape all the special characters enclosed in them except the dollar sign \$, the backquotes ‘ ‘ and the backslash \.
- Metacharacters used to redirect the input or output of a command. These are the characters <, >, >> and 2>. For further information, see the section on *Redirecting input and output*.
- Metacharacters entered at the end of a complete command to execute it and metacharacters used to separate a number of complete commands on one line. The characters that you enter at the end of a command to execute it are the newline character, entered by pressing the return key () and the ampersand &, which causes the command to be executed in the background. One of the most important characters used to separate a number of commands on the same line is the semicolon ;. Another important character in this group is the pipe |. Further information on its use is given in the section on *Filters, pipes and pipelines* in this chapter.

You will find more information on using shells, particularly the Bourne shell, in chapter 5 *The shell*. The metacharacters mentioned here and other metacharacters are covered in the description of your shell in the *Commands* manual [5].

Redirecting input and output

Most Reliant UNIX commands take their input from *standard input* (*stdin*). In Reliant UNIX, standard input is normally associated with the special file of your terminal, i.e. the entries you make are read from this special file.

The counterpart of standard input is *standard output* (*stdout*). In Reliant UNIX, standard output is also associated with the special file of your terminal, i.e. output is written to this special file. The kernel then causes this output data to be displayed on your screen.

In addition to standard input and standard output, Reliant UNIX features *standard error* (*stderr*), which is likewise associated with the special file of your terminal. Error messages issued by commands are written to this file and sent to the screen.

In Reliant UNIX, you can redirect standard input, standard output and standard error. One way of doing this is to use certain metacharacters to redirect them to other files. These characters are < (less than) for standard input, > and >> (greater than) for standard output and 2> for standard error.

Examples of redirection using these characters are given in section 5.2.2 *Redirecting input and output* of chapter 5 *The shell* and in the corresponding sections of the shell descriptions in the *Commands* manual [5].

Filters, pipes and pipelines

Reliant UNIX assigns standard input, standard output and standard error to each program when it is first started. These input/output channels are normally associated with the user's terminal. There are two ways of changing this assignment:

- Redirect one or more of these channels into a file. Basic information on this procedure is given in the section on *Redirecting input and output*.
- Make the standard output of one command the standard input of another.

Many Reliant UNIX commands are designed to read from standard input and write to standard output, i.e. they read from your keyboard and write to your screen by default. Using the alternatives described above, however, you can tell these commands to take their input from files or from other commands, write their output to files or pass their output to other commands. Commands that are capable of doing this are called *filters*.

One of the most important applications of filters is the *pipeline*. A pipeline is a sequence of at least two filters joined together using the *pipe* symbol, the vertical bar |. The advantage of a pipeline is that you do not have to execute each filter command individually, creating unnecessary temporary files. To solve a complex problem involving several stages, you can specify the commands in a pipeline. The pipeline automatically passes the results of one command to the next command for you.

In order to ensure that the pipeline is executed properly, you must specify the filters in the correct sequence. This means that a filter to the left of the pipe symbol must generate an output that the next command can process as its input.

Further information on filters, pipes and pipelines is given in section 5.2.3 *Filters and pipes* of chapter 5 *The shell* and in the corresponding sections of the shell descriptions in the *Commands* manual [5]. The command descriptions in these manuals also indicate which commands are filters.

Regular expressions

Regular expressions are character strings containing

- ordinary characters
- characters with special meaning

You can specify regular expressions when using Reliant UNIX commands that search for text matching a specific pattern. Once the text has been found, the command may, for instance, move the cursor to this position or replace the section of text with a different text. You can increase or decrease the set of character strings that match the pattern by specifying more or less characters of the pattern in the regular expression. A simple example: You want to search a text for the words *apple*, *supply*, *plane* and *diploma*. The only character string that these words have in common is *pl*. You can form a regular expression based on this character string in order to find all these words. There are two different ways of doing this. You can specify each word explicitly and nothing else, which is also a form of regular expression. Alternatively, you can specify a character string containing specific metacharacters that matches all the words you are searching for. The metacharacters that you would use in this example are the dot `.` and the asterisk `*`. The dot stands for any character and the asterisk stands for any number of this character. If you specify `.*pl.*` as a regular expression, you will find not only all the words specified above, but also all those character strings that contain *pl* at any position. The more characters you specify explicitly in the regular expression, the more you restrict the set of words that the command will find. If, for example, you specify `app.*`, you will only find *apple* and any other character strings that start with *app*. If you specify `.*ppl.*`, you will only find *apple*, *supply* and any other character strings that contain *ppl* at any position.

Not all Reliant UNIX commands can process regular expressions. A list of the commands that can is given in the *Commands* manual [5]. This manual (in two volumes) also contains a comprehensive table of the metacharacters for regular expressions and gives appropriate examples.

2.1.4 The Native Language System NLS

One of the main tasks of a program is to process data by interacting with users. It is important for this interaction to take place in the user's native language. Programs that meet this requirement make no assumptions about the language, about local conventions such as date and time formats, currency symbols or collating sequences, or about character sets, which can differ greatly in terms of size and representation. For example, Reliant UNIX was originally based on the ASCII character set, in which each character is coded as a unique sequence of 7 bits, thus limiting the total number of possible characters to 128. Many languages, however, contain more characters and call for 8-bit and even larger character sets.

Programs that make no assumptions about the language of the user and conform to local customs are known as *internationalized* programs. The data definitions of this internationalized environment are kept separate from the program logic. The technique of linking this environment to programs is known as *localization*.

The *Native Language System* (NLS) defines interfaces that make it possible to create internationalized programs. The following aspects of NLS are of particular interest to the user:

- NLS incorporates an announcement mechanism that enables users to set the language, the associated local customs and the character set according to their own requirements.
- NLS offers extended regular expressions that allow users to process characters specific to their personal language environment.
- NLS includes standard commands that can process character sets larger than accommodated by ASCII.

For further information on NLS, see the *Commands* manual [5].

2.2 Logging in

Before you can work with Reliant UNIX, you must log in. Logging in is similar to opening your office door: in order to open your door (login name), you require the appropriate key (password).

This section describes the login procedure.

2.2.1 Terminal

In order to log in, you must first switch on your terminal. The system's welcome screen (customized by the system administrator) is then displayed.

Note:

If only a blinking cursor is displayed instead of the welcome screen once you have switched on the terminal, press `↵` or `END`.

A terminal is an input/output device; you use it to pass instructions to the Reliant UNIX system, and Reliant UNIX uses it to output data.

Every terminal must be configured properly before data can be exchanged with the Reliant UNIX system. Since terminal configuration is the responsibility of the system administrator, this manual does not describe it in further detail. For more information, see the *System Administrator's Guide* [8]. If you have problems with the terminal when logging in, consult your system administrator.

2.2.2 Login name

You require a login name before you can log in. The login name informs Reliant UNIX whether you are authorized to use the system. If you do not yet have your own login name, consult your system administrator, who is responsible for setting up login names.

You enter your login name in response to the *login:* prompt each time you log in. You then press `↵`. You correct input errors using the backspace key `⌫` (for information on correcting errors, see section 2.3.2 *Keyboard input*).

In the following example, the user logs in under the login name *darkstar*:

```
login: darkstar
```

2.2.3 Password

In order to prevent unauthorized access to data, each login name is protected with a password. The system administrator defines the password when setting up the login name. If the system administrator has not defined a password for your login name, you can assign a password yourself (see the section on *Assigning or changing a password*). You enter your password in response to the *password:* prompt. For security reasons, the password you enter does not appear on the screen.

The following example illustrates the login procedure for the login name *darkstar*. The login name is protected by the password *Rossini!*. Note that the password entered does not appear on the screen.

```
login: darkstar
password: Rossini!
```

Once you have entered the correct password, the shell prompt, usually the dollar sign (\$), appears on the screen, indicating that you have established contact with Reliant UNIX. Reliant UNIX now expects you to enter a command.

If you entered the password incorrectly, you receive the error message *Login incorrect* and Reliant UNIX displays the *login:* prompt again.

Rules for assigning passwords

You must observe the following rules when defining passwords:

- The system administrator determines the minimum number of characters for a password (by defining the *PATHLENGTH* variable in the */etc/default/passwd* file). If you use a password with more than eight characters, the system ignores all but the first eight characters.
- A password must contain at least two alphabetic characters and at least one number or special character. You can use both lowercase and uppercase letters.

Assigning or changing a password

You should change your password at regular intervals in order to ensure the security of sensitive data. To do this, you use the *passwd* command.

In the following example, the password for the login name *darkstar* is changed from *Rossini!* to *%Sevilla*. Once again, note that the passwords entered do not appear on the screen.

```
$ passwd
passwd: Changing password for darkstar
Old password: Rossini!
New password: %Sevilla
Re-enter new password: %Sevilla
$
```

Table 1 summarizes the syntax and function of the *passwd* command.

Command summary	
Command	Argument
passwd	<i>login_name</i>
Function:	<i>passwd</i> assigns a new password or changes the existing password for a login name.
Argument: <i>login_name</i>	Login name for which you want to assign or change the password.
Comments:	Users without system administrator privileges can only modify their own password.
Full syntax: passwd [<i>-s</i> [<i>-a</i>]] [<i>-l</i>] [<i>-d</i>] [<i>-f</i>] [<i>-n</i> <i>minimum</i>] [<i>-x</i> <i>maximum</i>] [<i>-w</i> <i>warning</i>] [<i>login_name</i>]	

Table 1: Summary of the *passwd* command

2.2.4 Problems during login

If you have problems during the login procedure, check whether you have observed the following rules:

- You can only use the backspace key , not the arrow keys  , to correct your input.
- You must make sure that  (shift lock) has not been pressed, since Reliant UNIX is case-sensitive.

If you are still unable to solve the problem, consult your system administrator.

Table 2 summarizes the steps involved in the login procedure.

1. Switch your terminal on. If the welcome screen does not appear, press  or 
2. Enter your login name after the <i>login:</i> prompt and press 
3. Enter your password after the <i>password</i> prompt and press 
4. Reliant UNIX responds by displaying the prompt. The default prompt is the dollar sign (\$).

Table 2: Login procedure

2.3 Communicating with the system

Once you have completed the login procedure successfully and the \$ prompt is displayed on the screen, you have set up a connection with the Reliant UNIX operating system.

2.3.1 The shell

Once you have logged in, you are connected with the shell, or command interpreter. This is a type of program architecture in which a program forms a shell around the operating system kernel. The shell is the program that links you, the user, directly with a larger number of programs, acting as a direct interface between you and the computer. For this reason, you only come into contact with the shell and its commands. You do not have to consider the internal working of the computer since the shell handles all other programs.

The shell provides the link between all users and the kernel of the operating system. Users are given a working environment and can use the computer in parallel. The shell receives and interprets input, either passing it to the operating system kernel or performing the corresponding operations itself. Reliant UNIX users do not have direct access to the operating system kernel.

For further information on the shell, see chapter 5 *The shell*.

2.3.2 Keyboard input

As soon as the \$ prompt appears on the screen, the shell is ready to accept your instructions.

You can now pass your input to Reliant UNIX via the keyboard. Unless otherwise specified, you must enter all Reliant UNIX commands in lowercase.

Note:

When typing in commands, bear in mind that Reliant UNIX makes a distinction between lowercase and uppercase letters.

As soon as you enter an instruction via the keyboard, your input is displayed on the screen. This gives you the impression that your input is sent directly from the keyboard to the screen. In reality, however, it follows a different route. Each character you enter via the keyboard is sent directly to the computer. The computer sends it back to the screen and displays the characters to enable you

to check your input. When you terminate your input by pressing the return key , the computer already has your input and does not have to read it from the screen.

The computer does not process your input until you indicate that it is complete by pressing . If you have not yet pressed , you can correct your input.

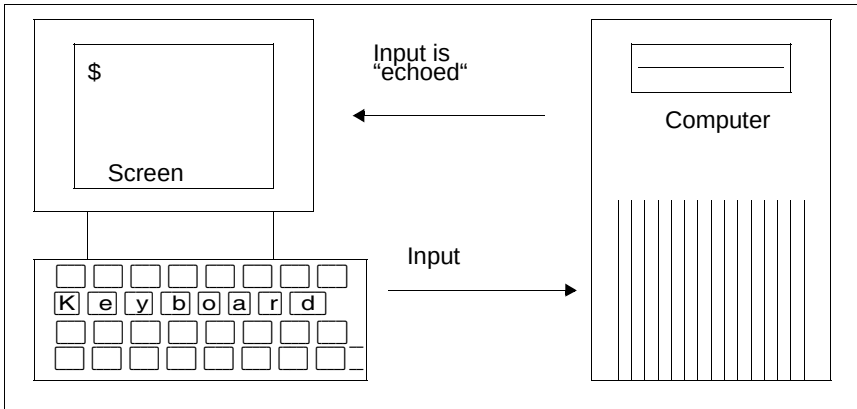


Figure 1: What happens to your input

Correcting input errors

Since your input is sent to the computer immediately, you can only correct it using special keys and combinations of keys.

The following options are available for correcting input:

- The backspace key  deletes the character to the left of the cursor. You can use this key to consecutively delete one or more of the characters you have just entered on the current line.
- The key combination  h deletes the character to the left of the cursor. You can use this key combination to consecutively delete one or more of the characters you have just entered on the current line. The key combination  h has the same effect as the backspace key .
- The  key deletes the current line in its entirety, cancelling the entire input. The \$ prompt appears once again on a new line.
- The key combination  c deletes the current line in its entirety.

It is not possible to correct input using the arrow keys  and . If you do press the arrow keys, the corresponding control characters are sent to the computer. Reliant UNIX interprets these control characters not as a correction, but as part of the input. What you see on the screen appears to be correct because the cursor moves to the desired position when you press the arrow keys. In reality, however, your input consists of

- the characters displayed on the screen
- the control characters for the arrow keys
- the characters you have overwritten

If you now terminate your input by pressing , the input you send to the computer is different to the input you can see on the screen.

The next example illustrates how to correct your input under Reliant UNIX. The *date* command outputs the current date and time. First, the user attempts to correct the input using . Because Reliant UNIX interprets  as a control character, it looks for a command with a name that it does not recognize. The user then corrects the input properly using .

```
$ datr  e
datr^[[De: not found
$ datr  e
Mon Feb 10 18:06:00 MET 1992
$
```

Table 3 summarizes the important keys and key combinations.

Key or key combination	Function
 h	Delete a character on the current command line
	Delete a character on the current command line
 c	Delete the current command line
	Delete the current command line
 m	Terminate the input line
	Terminate the input line
	Terminate the input line
 d	Log out
	Log out

Table 3: Important keys and key combinations

Entering commands

If you want to have the computer solve a problem, you must instruct it to perform a specific task. You do this by entering a command.

A command is a file containing instructions for the computer in a command language. A wide range of commands is available to Reliant UNIX users. Detailed information on the Reliant UNIX user commands is given in the *Commands* manual [5].

Once you have logged in, you can enter a command at any time. When the \$ prompt appears on the screen, the system is ready to receive a command. As soon as you have entered a command and pressed `[↵]`, the command is released and executed.

You enter Reliant UNIX commands as follows:

- The \$ prompt is normally followed by a blank.
- You can enter a command directly after the blank.
- You terminate command input by pressing `[↵]`.

Each Reliant UNIX command is contained in a separate file (with the exception of shell built-ins. For further information on *shell built-ins*, see the *Commands* manual [5] under the entry *Shell, The shell as command interpreter*).

The names of the command files are entered in directories, e.g. */usr* and */usr/sbin*. When you issue a command, the shell reads the command name, looks for the corresponding file name in these directories and executes the command you have entered.

Once the command has been executed, Reliant UNIX outputs the \$ prompt again.

The following example illustrates how to enter the Reliant UNIX *who* command. The *who* command lists the login names of all users who are currently logged in, the number of the terminal at which each user is logged in and the login time.

```
$ who
darkstar      tty002      Feb  10 09:12
arthur2       tty004      Feb  10 08:06
james         tty007      Feb  10 11:55
tsv           tty018      Feb  10 06:00
$
```

The \$ prompt that follows the command output indicates that Reliant UNIX is ready to process a new command.

Further information on command processing is given in chapter 5 *The shell*.

2.4 Logging out

There are three ways of logging out:

- Enter the *exit* command.
- Press the key combination **CTRL** d.

Before doing so, make sure that the input line is empty, i.e. that it contains only the \$ prompt (if necessary, delete characters using **ⓧ** only).

- Press **END**.

Before doing so, make sure that the input line is empty.

After a short while, the welcome screen will be displayed again. This indicates that you have logged out successfully.

If, instead of the welcome screen, the \$ prompt reappears, you are in what is known as a subshell (see chapter 5 *The shell*). If this occurs, simply log off again.

Caution:

If you simply switch off the terminal, you do not actually log out unless your terminal setting has been modified appropriately (by means of the *stty* command and the *hupcl* option. For further information, refer to the *Commands* manual [5]). If you do not log out properly, the connection with Reliant UNIX is retained and unauthorized users can access the system by switching on your terminal.

2.5 Summary

1.	Switch your terminal on. If no welcome screen appears, press ↵ or END .
2.	Enter your login name after the <i>login:</i> prompt and press ↵ .
3.	Enter your password after the <i>password:</i> prompt and press ↵ .
4.	Reliant UNIX responds by displaying the prompt. The default prompt is the dollar sign (\$).

Table 4: Summary of the login procedure

Key or key combination	Function
CTRL h	Delete a character on the current command line
ⓧ	Delete a character on the current command line
CTRL c	Delete the current command line
DEL	Delete the current command line
CTRL m	Terminate the input line
↵	Terminate the input line
MENU	Terminate the input line
CTRL d	Log out
END	Log out

Table 5: Important keys and key combinations

3 The file system

3.1 Introduction

In order to use Reliant UNIX effectively, it is essential for you to have a knowledge of the file system. This chapter introduces you to the file system, describing its structure and showing you how to use the file system itself and the files it contains.

The first section in this chapter, *The structure of the file system*, gives general information on the file system.

The subsequent sections cover important Reliant UNIX commands for manipulating the file system. Specifically, these are commands used to create your own directory structure and access files and directories.

Each command is described in a separate subsection. Each subsection ends with a table summarizing the syntax and function of the command involved.

The commands listed here are basic to an efficient use of the file system. Some of these commands have additional, more complex applications beyond the scope of this manual. These commands are described in detail in the *Commands* manual [5] or in other Reliant UNIX manuals which are cross-referenced where appropriate.

3.2 The structure of the file system

The file system forms the basis for the management of information. In it, your data is stored in structured form so that you can access it as required.

The Reliant UNIX file system consists of files and directories. A file is a container for coded information in which data is stored. A directory can contain files and other directories.

The set of all files and directories is organized in a hierarchical structure that branches downward. Fig. 2 shows a sample Reliant UNIX file hierarchy.

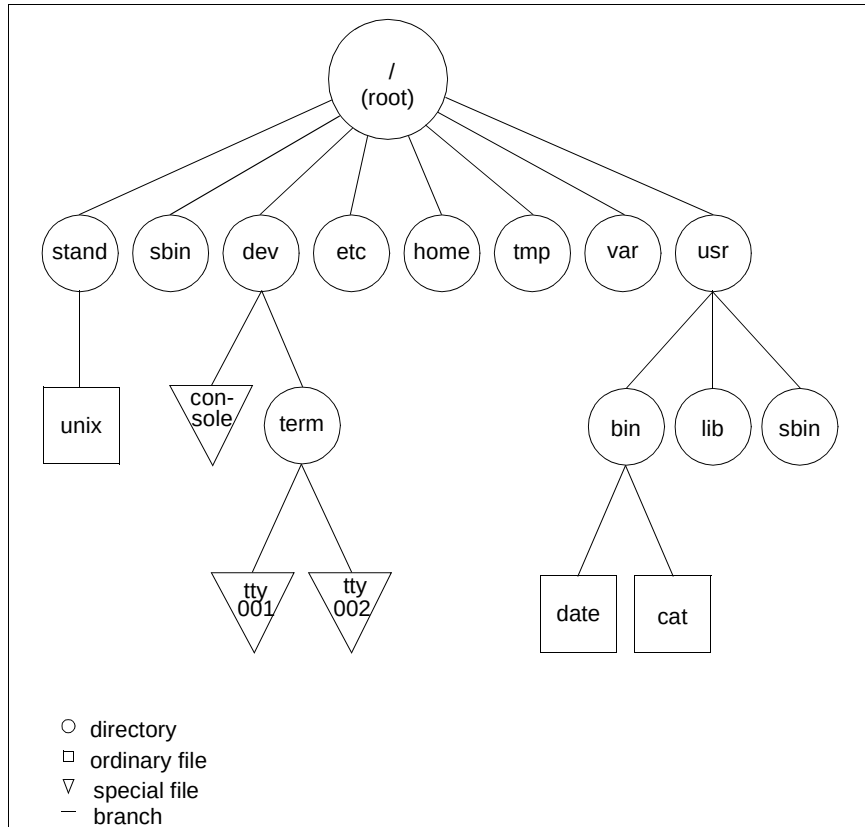


Figure 2: Sample file system

3.2.1 Files

A file is a container in which data is stored and made accessible. It may contain text for a report, for example, or code for a program.

In Reliant UNIX, there are no fixed rules about the file structure. Each file consists of a sequence of coded characters. A file contains no information about the file itself. The way in which the contents of a file are evaluated always depends on the program that is using the file. The Reliant UNIX commands themselves are also stored in files.

Reliant UNIX does not place any restrictions on the maximum file length or the number of files that a file system can contain. In practice, however, restrictions are imposed by the capacity of the storage medium and various other factors.

For information on creating files, see section 3.3.3 *Creating files using the cat and echo commands*.

File names

Each file is identified uniquely by its file name. Reliant UNIX file names are subject to few restrictions. You should, however, observe the following conventions:

- You can use all characters except the slash (/) (the slash (/) acts as a delimiter between files and directories in path names. For further information on path names, see section 3.2.6 *Path names*).
- It is advisable not to use certain characters and keys. These are
 - the space bar
 - the tab key
 - the backspace key

and the following characters:

- ? @ # \$ ^ & * () ' [] \ | ; ' " < >

If the name of a directory or file contains a blank or a tab character, you must enclose the name in double quotes on the command line.

- Do not use + or - as the first character of a file name.
- If you use the dot . as the first character of a file name, the file is ignored by the *ls* command unless you enter *ls* with the option *-a*.
- Reliant UNIX makes a distinction between uppercase and lowercase letters. For example, Reliant UNIX considers the files *project*, *PROJECT* and *Project* to be three different files.
- There are no significant restrictions regarding the length of file names (the maximum length of a file name is 255 characters). For practical reasons, however, you should choose a short, informative file name.

The following are examples of valid names for files or directories:

memo	MEMO	section2	ref:List
file.d	chap3+4	item1-10	schedule

3.2.2 Directories

Reliant UNIX allows you to group files together on the basis of any criteria you choose. The file names are written to a special type of file containing only organizational information on the files. These files are called directories. As well as file names, directories can contain the names of other directories (known as subdirectories). These subdirectories can in turn contain several more files and/or directories.

These directories and files form a multilevel, hierarchical file system organized according to user-specific criteria. Fig. 2 illustrates the hierarchical structure of the Reliant UNIX file system. In Reliant UNIX, there is no limit to the number of directory levels you may create. For the sake of clarity, however, you should not create an excessive number of directory levels.

Directories are subject to the same naming conventions as files, as described in section 3.2.1 *Files*. For further information on creating and deleting directories, see sections 3.3.1 *Creating directories: the mkdir command* and 3.3.5 *Deleting directories: the rmdir command*.

You can list the contents of a directory using the *ls* command. This command outputs the names of the files and directories contained in the directory you specify. For further information, see section 3.3.2 *Listing the contents of a directory: the ls command*.

3.2.3 The hierarchical structure of the file system

As you have already seen, the Reliant UNIX file system has a hierarchical structure that branches downward. This manual refrains from comparing the Reliant UNIX file system with an "inverted tree that grows downward", however (in the hope that trees will continue to grow upwards in the future).

Instead, refer once again to Fig. 2 to obtain a clear picture of how the Reliant UNIX file system is structured:

The source of the file system is a directory called *root*. This directory is represented by the slash (/). If you move down to the next level of the file hierarchy, you come to what are known as the system directories.

The system directories form the first level at which files and directories are organized in subject groups. Reliant UNIX predefines the format of the first level. Fig. 2 shows some important system directories, e.g. *dev*, *usr*, *sbin* and *home*.

The data stored in the system directories contains important information required for running the operating system. The system directory *dev*, for example, contains the special files, *usr* contains Reliant UNIX commands and system data while *sbin* contains the Reliant UNIX system administrator commands.

Since only the system administrator can affect the files and directories contained in the system directories, these are not covered in more detail in this manual. For further information, see the *System Administrator's Guide* [6].

There is one special system directory, however, that needs to be mentioned here. The system directory *home* contains the user data; in other words, it groups together the directories of all users registered in the Reliant UNIX system (for further information on the login directory, see section 3.2.4 *The login directory*). In the file system illustrated in Fig. 3, the directories of the login names *darkstar*, *arthur2* and *tsv* are grouped together under *home*.

From the uppermost level, the directory structure extends further downward. Since there is no limit to the number of files and directories you can create, you can organize your file system in a variety of different ways.

The hierarchical file structure in Reliant UNIX provides the users of the operating system with a number of options:

- In theory, each user can work with any file or directory throughout the file system (provided that permission to access the file or corresponding directory has been granted. For further information on permissions, see section 3.2.8 *Permissions for files and directories*.)
- Several users can access the same file. This is particularly useful for a team of people working on a single project, for example.
- Users can organize their files to meet their own requirements by grouping them together into directories according to subject area.
- A file can be stored under the same name in a number of different directories.

3.2.4 The login directory

Once you have logged in successfully, the Reliant UNIX system places you at a specific point in its file system structure called the login directory. The name of your login directory is usually the same as your login name (the system administrator assigns both these names to you when setting up your Reliant UNIX system account).

The system administrator also defines the location of your login directory as a subdirectory in the Reliant UNIX file system hierarchy. This is usually under the system directory *home*.

Fig. 3 shows the position of the system directory *home* in relation to the directories *darkstar*, *arthur2* and *tsv*, and to other important Reliant UNIX system directories.

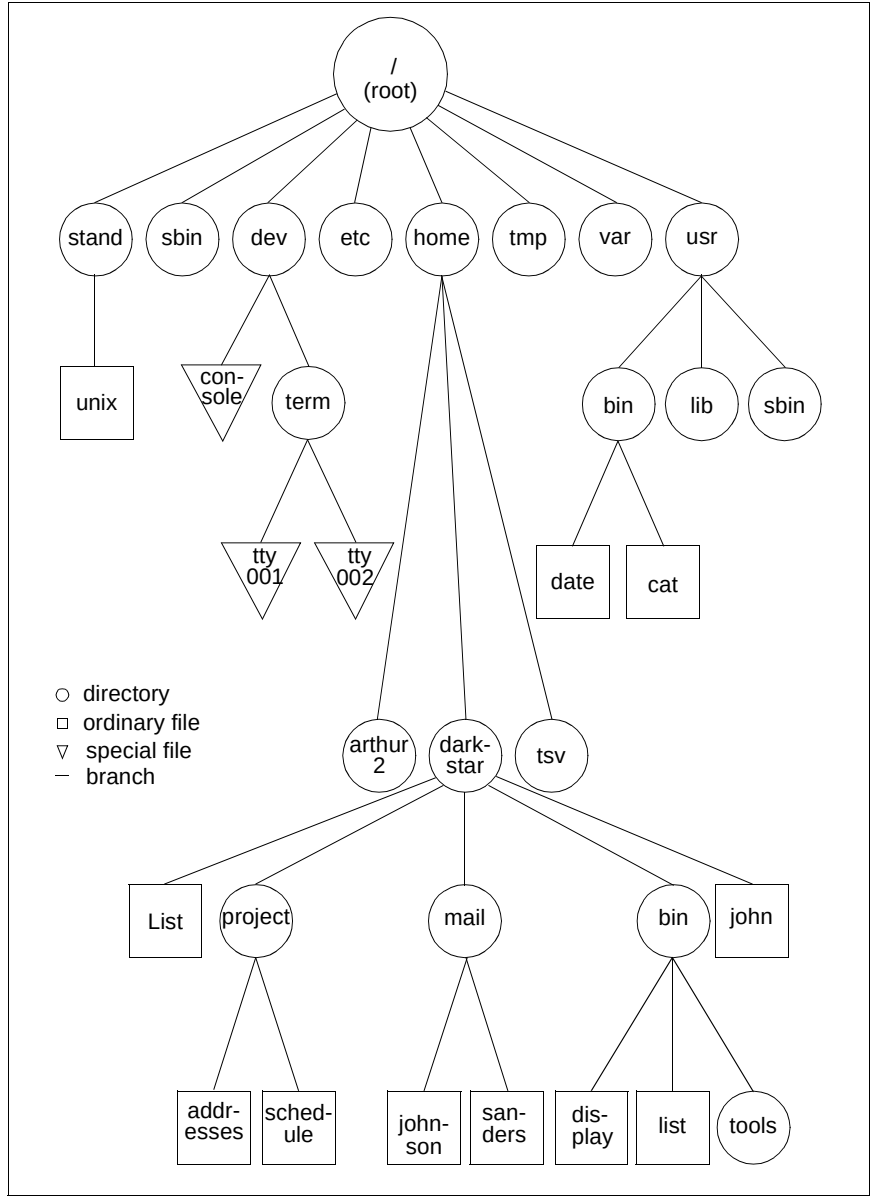


Figure 3: The directories under home

Within your login directory, you can create files and additional directories (subdirectories), that can in turn contain files and directories. You have full responsibility for everything you create in your login directory because you are its owner. You can move and delete your files and directories and determine which users are allowed to access them (for further information, see section 3.2.8 *Permissions for files and directories*).

The home directory is normally identical to the login directory. The path name of the home directory is defined in the *HOME* variable. You can display the contents of the *HOME* variable using the *echo \$HOME* command. The home directory automatically becomes the current directory if you enter the *cd* command on its own (further information on the *cd* command is given in section 3.2.7 *Moving to another directory: the cd command*).

3.2.5 The current directory

Your current directory is the directory in which you are currently working (also called the working directory). When you move to another directory using the *cd* command, that directory becomes your new current directory (for further information on the *cd* command, see section 3.2.7 *Moving to another directory: the cd command*).

The Reliant UNIX command *pwd* (short for **p**rint **w**orking **d**irectory) displays the name of the current directory. If, for example, you log in under the login name *darkstar* and enter the *pwd* command the first time the prompt is displayed, the system will respond as follows:

```
$ pwd
/home/darkstar
$
```

The system response gives you both the name of the directory in which you are working (*darkstar*) and the location of that directory in the file system. The path name */home/darkstar* tells you that the root directory (represented by the first slash (/)) contains the *home* directory, which in turn contains the subdirectory *darkstar*. The first slash in the name stands for the root directory; all other slashes are used to separate the names of directories and files and to show the position of each directory relative to the root directory.

A name that describes the location of a directory or file in this way is known as a path name. The next section deals with path names in Reliant UNIX.

Table 6 summarizes the syntax and function of the *pwd* command.

Command summary	
Command	
pwd	
Function:	<i>pwd</i> displays the absolute path name of the directory in which you are currently working.
Full syntax:	
pwd	

Table 6: Summary of the *pwd* command

3.2.6 Path names

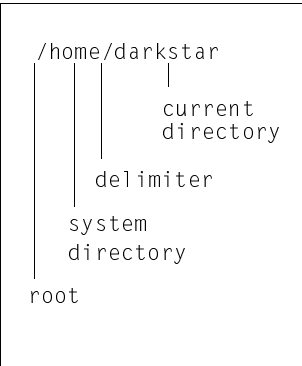
In a hierarchical file system, it must be possible to pinpoint the exact location of each file and directory. In Reliant UNIX, the path name of a file or directory shows you the route you must follow to find it. The path name shows the location of the file or directory within the file system and provides directions for reaching it. Knowing how to follow these directions is your key to moving around the file system successfully. The first step is to learn about the two types of path name: absolute and relative.

Absolute path name

An absolute path name gives directions that start in the root directory and lead you down through one or more directories to a particular file or directory.

Because an absolute path name always starts at the root directory, its first character is always a slash (/). The final name in an absolute path name can be the name of a file or of a directory. All other names in the path must be directory names.

The following example illustrates how an absolute path name is constructed. The key below it describes the individual components of the absolute path name */home/darkstar*.



<code>/</code> (first character)	The slash that appears as the first character in the path name represents the root of the file system.
<code>home</code>	The system directory one hierarchy level below <i>root</i> to which <i>root</i> branches.
<code>/</code> (not first character)	The next slash acts as a delimiter between the directory names <i>home</i> and <i>darkstar</i> .
<code>darkstar</code>	The current directory.

Follow the dashed lines in Fig. 4 to trace the absolute path to the `/home/darkstar` directory.

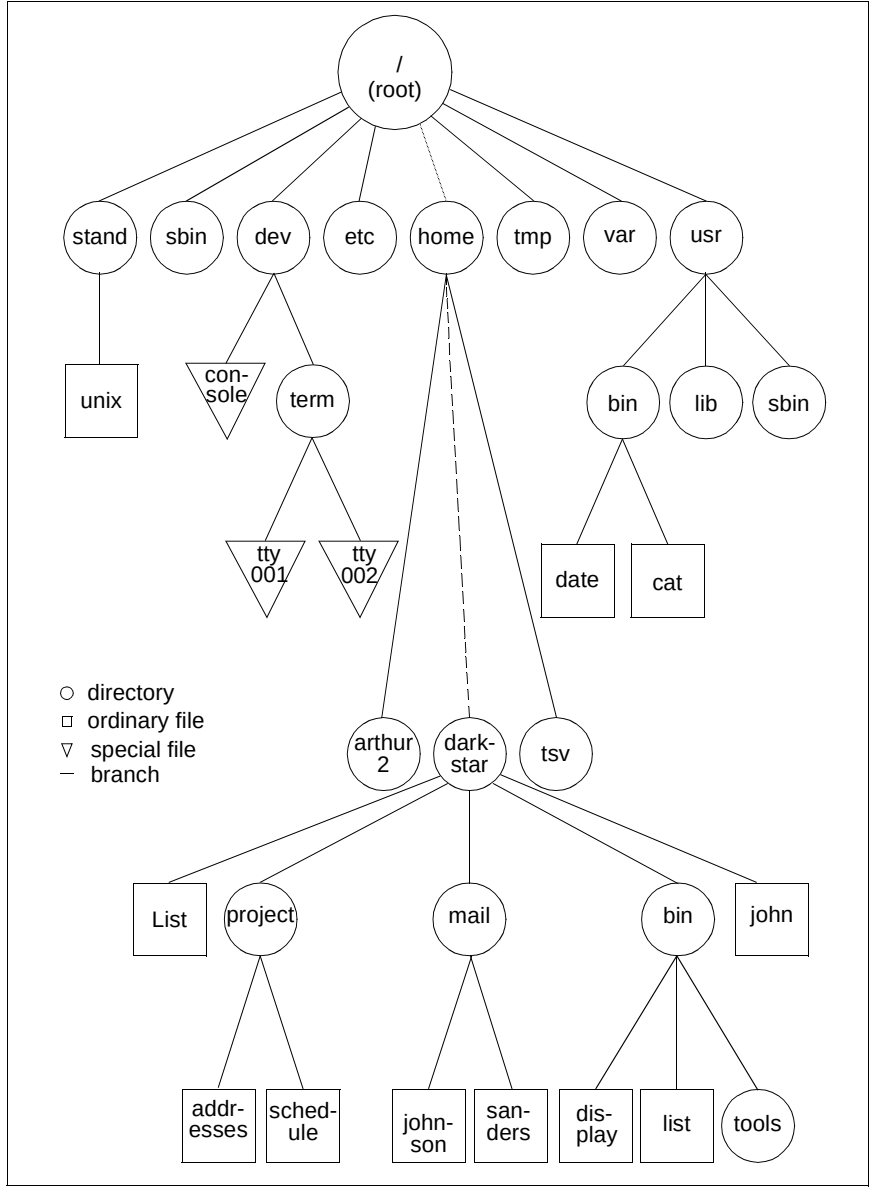


Figure 4: Absolute path name of the /home/darkstar directory

Relative path name

A relative path name gives directions that start in your current directory and lead you through one or more directories either above or below the current directory in the file hierarchy to a particular file or directory.

By moving down from your login directory, you can access your own files or directories. By moving up from your login directory, you eventually reach the root directory. From there you can move anywhere in the file system.

A relative path name begins with one of the following: a directory name, a file name, a dot (.) or two dots (..). The single dot (.) stands for your current directory. Because the relative path name always gives directions starting from the current directory, you can prefix any relative path name with the dot (.), although this is not mandatory.

Two dots (..) represent the directory immediately above your current directory in the file system hierarchy.

The final name in a relative path name can be the name of a file or of a directory. All other names in the path, however, must be directory names.

The following examples show you how a relative path name is structured and how it points the way to a file or directory. For example, suppose that you are working in the *darkstar* directory in the sample file system (see Fig. 3). *darkstar* contains the directories named *project*, *mail* and *bin* and the files named *List* and *john*. The relative path name to any of these is simply the name of the file or directory involved, e.g. *project* or *john*.

The dashed line in Fig. 5 traces the relative path from *darkstar* to *project*.

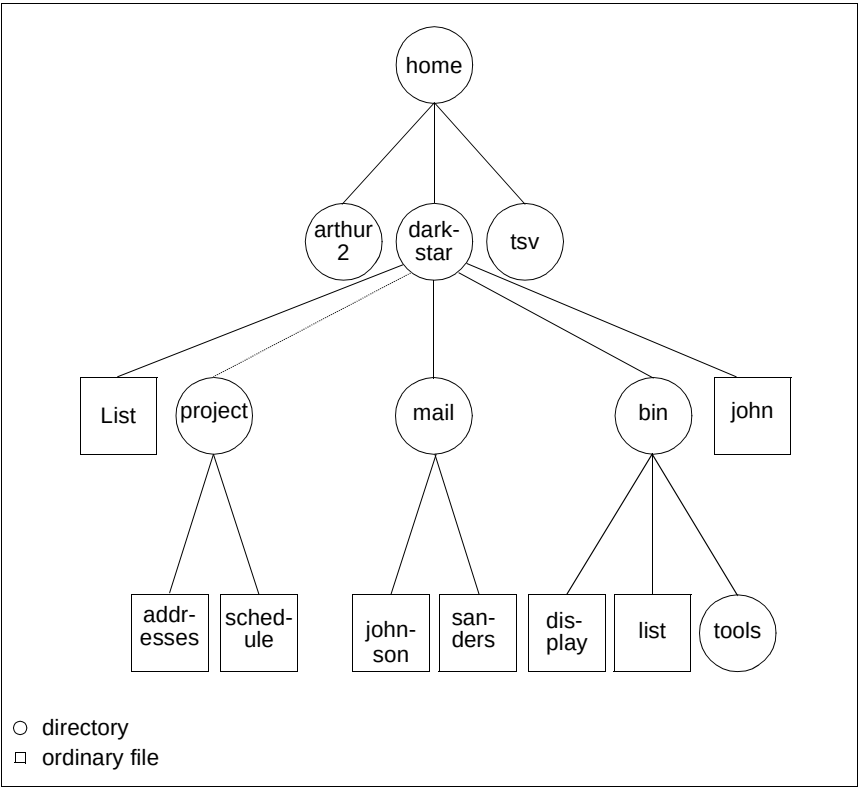


Figure 5: Relative path name from *darkstar* to *project*

The *project* directory, a subdirectory of *darkstar*, contains the files *schedule* and *addresses*. The relative path name from *darkstar* to the *addresses* file is *project/addresses*.

The dashed line in Fig. 6 traces this relative path. Note that the slash (/) in this path name separates the *project* directory from the *addresses* file. Here, the slash (/) acts as a delimiter, indicating that the *addresses* file is subordinate to the *project* directory.

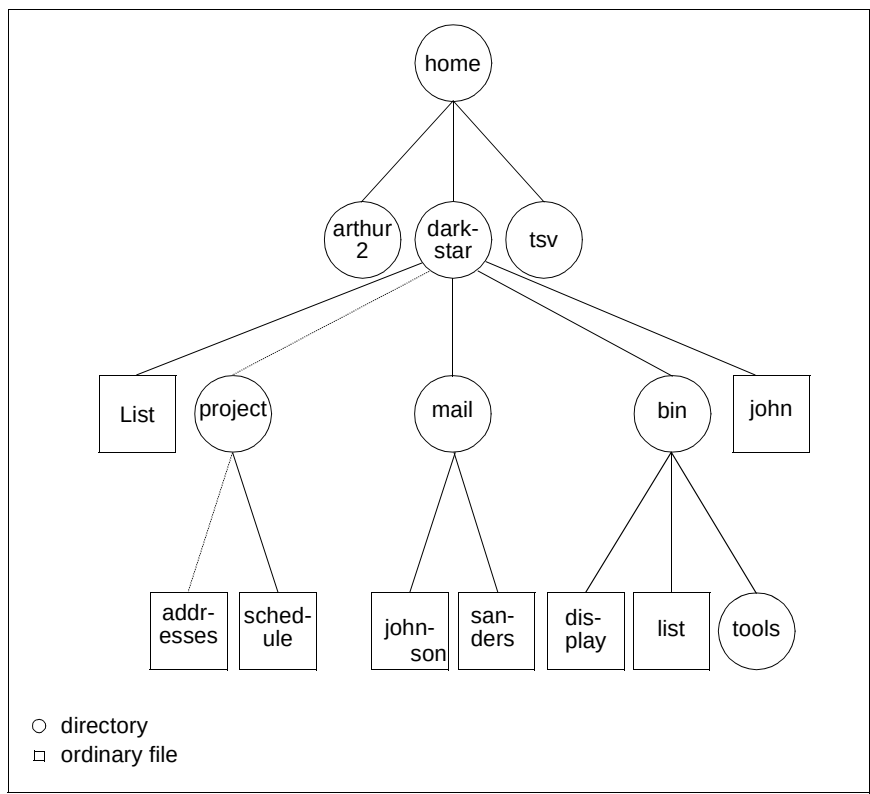


Figure 6: Relative path name from darkstar to addresses

The above examples showed you how to specify a path to a file that is located below your current directory. However, you can also move upwards in the file hierarchy.

You use the relative path name `..` to specify the parent directory of your current directory.

If, for example, you are in the *project* directory of the sample file system (Fig. 3), `..` is the relative path name to the *darkstar* directory and `../..` is the path name to *home*, the parent directory of the *darkstar* directory.

From the *project* directory, you can also trace a path to the file named *sanders* which is located in the *mail* directory. The path name in this case is *../mail/sanders*. *..* moves you up to the parent directory *darkstar* from where you move back down in the file hierarchy to the *mail* directory that contains the *sanders* file.

Bear in mind that you must specify an absolute path name and not a relative path name if you want the path specification not to be relative to the current directory. Apart from this restriction, you are free to specify either type of path name.

Table 7 contains examples of relative and absolute path names.

Path name	Meaning
/	Absolute path name of the root directory
/usr/bin	Absolute path name of <i>bin</i> . <i>bin</i> is located in <i>usr</i> , a subdirectory of <i>root</i> .
/home/darkstar/bin/tools	Absolute path name of <i>tools</i> ; <i>tools</i> is located in the <i>bin</i> directory, which is a subdirectory of the <i>darkstar</i> directory; <i>darkstar</i> is a subdirectory of the home directory, which in turn is a subdi- rectory of <i>root</i> .
bin	Relative path name of <i>bin</i> ; <i>bin</i> is located in the current directory. If the current directory is <i>darkstar</i> , the relative path name <i>bin</i> corresponds to the absolute path name <i>/home/darkstar/bin</i> . If the current directory is <i>usr</i> , the relative path name <i>bin</i> corresponds to the absolute path name <i>/usr/bin</i> .
bin/tools	Relative path name of <i>tools</i> ; <i>tools</i> is located in the <i>bin</i> directory. If the current directory is <i>darkstar</i> , the relative path name <i>bin/tools</i> corresponds to the absolute path name <i>/home/darkstar/bin/tools</i> .
tools	Relative path name of <i>tools</i> ; <i>tools</i> is located in the current directory.

Table 7: Sample path names

3.2.7 Moving to another directory: the `cd` command

As you have already seen, when you first log in you are placed in your login directory. This remains your current directory until you move to another directory. To move to another directory, you enter the `cd` command (short for **change directory**) with the path name of the second directory. This then becomes your current directory.

```
cd [directory]
```

You can specify any valid path name (absolute or relative) as an argument for the `cd` command. If you do not specify a path name, `cd` will move you to your home directory.

If, for example, you want to move from the *darkstar* directory to the subdirectory *project* in the sample file system (Fig. 3), you enter the `cd project` command line. You can then verify your new location in the file system using the `pwd` command:

```
$ cd project
$ pwd
/home/darkstar/project
$
```

The `cd` command accepts both absolute and relative path specifications. If, for example, you want to move from the *project* directory to the *mail* directory, you can enter either of the following command lines:

```
$ cd /home/darkstar/mail
$ cd ../mail
```


Table 8 summarizes the syntax and function of the `cd` command.

Command summary	
Command	Argument
<code>cd</code>	<i>directory</i>
Function:	<i>cd</i> changes your position in the file system from the current directory to a directory you specify. If you do not specify a directory name as an argument, the <i>cd</i> command moves you in your home directory.
Argument: <i>directory</i>	Name of the directory you want to be your current directory.
Comments :	To access a directory that is not in your current directory, you must use the relative or absolute path name in place of the directory name.
Full syntax: cd [<i>_directory</i>]	

Table 8: Summary of the `cd` command

3.2.8 Permissions for files and directories

Because Reliant UNIX is a multiuser system, you are not normally the only person using the file system. The different system users can access other directories via path names and work with data that other users have stored there. This has its advantages, as you have already seen, but it can also create security problems. Without some form of protection, any user could alter the entire file system.

For this reason, Reliant UNIX provides various protection mechanisms that protect files and directories against unauthorized access:

- Each user has a unique login name
- Each login name is protected by a password
- Each file and directory is protected by permissions

You have already become familiar with the login name and password in chapter 2 *Basics for Reliant UNIX system users*. You will now learn about permissions.

Permissions in Reliant UNIX

Every file or directory created in Reliant UNIX is assigned default permissions (protection bits). Permissions define

- who is allowed to read and copy files and directories
- who is allowed to modify files and directories
- who is allowed to execute files and directories

The different permissions are represented by the following three letters:

- r* for read permission
- w* for write permission
- x* for execute permission

These permissions are interpreted as follows for files and directories:

Permission	File	Directory
read	Allows system users to read a file and copy its contents.	Allows system users to read the contents of a directory (using the <i>ls</i> command).
write	Allows system users to modify a file.	Allows system users to delete and create directory entries.
execute	Allows system users to run an executable file.	Allows system users to move to a directory (if <i>w</i> is not set) or to work in a directory (if <i>w</i> is set).

Table 9: The meaning of permissions for files and directories

There are three other permissions in addition to the permissions *r*, *w* and *x* described above: the *s* bit (**s**et user ID or **s**et group ID), the *l* bit (**l**ocking bit) and the *t* bit (**s**ticky bit).

However, the special meaning of these three permissions is of interest only to programmers and system administrators (for further information, see the *System Administrator's Reference Manual* [7]).

The categories of Reliant UNIX users

In Reliant UNIX, there are also three categories of user to which you can grant or deny these permissions separately. Each of these categories is represented by a letter:

- u** (user) You, the owner of your files and directories.
- g** (group) Members of the group to which you belong (e.g. people working on the same project or members of a department). The system administrator assigns you to this group when setting up your account.
- o** (other) All other system users.

Listing the current permissions

You can determine what permissions are currently in effect for a file or directory using the `ls -l` command. This command produces a long listing of a directory's contents (only the part of the output that is relevant to permissions is given below. For further information on the `ls` command, see 3.3.2 *Listing the contents of a directory: the ls command*). If you enter the `ls -l` command while in the `/home/darkstar/bin` directory in the sample file system (Fig. 3), you receive the following output:

```
$ ls -l
total 35
-rwxr-xr-x  1 darkstar  phoenix  9346 Nov  1  08:06  display
-rw-r--r--  1 darkstar  phoenix  6428 Dec  2  10:24  list
drwx--x--x  2 darkstar  phoenix   32 Nov  8  15:32  tools
$
```

The permissions for the files *display* and *list* and for the *tools* directory are listed on the left side of the screen:

```
-rwxr-xr-x (for the display file)
-rw-r--r-- (for the list file)
drwx--x--x (for the tools directory)
```

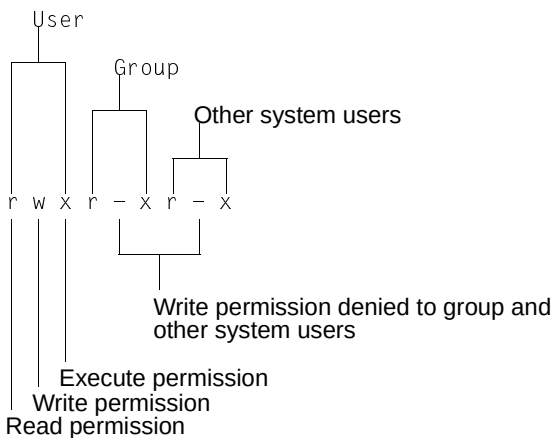
The first character identifies the file type (for further information on file types, see section 3.3.2 *Listing the contents of a directory: the ls command*). The following nine characters specify the permissions for the three categories:

- The second, third and fourth characters specify the permissions of the file owner.
- The fifth, sixth and seventh characters specify the permissions of the group.

- The eighth, ninth and tenth characters specify the permissions of all other system users.

The characters *r*, *w* and *x* show what permissions are granted to these user categories. A hyphen (-) instead of *r*, *w* or *x*, indicates that read, write or execute permission has not been granted.

The following diagram shows the permissions set for the *display* file:



As you can see, the owner has read permission (*r*), write permission (*w*) and execute permission (*x*). Group members and other system users, however, have only read permission (*r*) and execute permission (*x*).

The output of the *ls -l* command also shows you the owner of the file and the name of the user group. In this example, the user *darkstar* is the owner of the files and the name of the group is *phoenix*.

Changing the permissions for files

If you are the owner of a file, you can change the existing permissions using the *chmod* command.

The *chmod* command has the following standard syntax:

`chmod who+permission file(s)`

`chmod who-permission file(s)`

`chmod who=permission file(s)`

The components of these command lines have the following meaning:

Component	Meaning
who	One of the three user categories (u, g or o): u = user g = group o = other system users a = all three user categories
+ or -	Instruction that grants (+) or denies (-) permissions
=	Instruction that grants "absolute" permission, i.e. the specified permissions are granted and all others are denied.
permission	Any combination of the three permissions: r = read w = write x = execute
file(s)	The specified file or directory name(s). You can also specify absolute or relative path names here.

Make sure that you do not type a blank between *who*, + (or !=) and the *permission* on the *chmod* command line.

The following example illustrates how you, the owner of the *display* file, can protect this file from being changed accidentally by denying yourself write permission. You can then use the *ls -l* command to verify that the permission has changed:

```
$ chmod u-w display
$ ls -l
total 35
-r-xr-xr-x  1 darkstar    phoenix    9346 Nov 1  08:06 display
-rw-r--r--  1 darkstar    phoenix    6428 Dec 2  10:24 list
drwx--x--x  2 darkstar    phoenix     32 Nov 8  15:32 tools
$
```

The output of the *ls -l* command shows you that the members of the group *phoenix* and all other system users are denied write permission for the *display* file. They do have read permission, however, which enables them to copy the file into their own directory and write changes to it. To prevent the members of your group and all other system users from doing this, you can deny them read permission for the file, as shown in the next example:

```
$ chmod go-r display
$ ls -l
total 35
-r-x--x--x  1 darkstar      phoenix    9346 Nov  1  08:06 display
-rw-r--r--  1 darkstar      phoenix    6428 Dec  2  10:24 list
drwx--x--x  2 darkstar      phoenix     32 Nov  8  15:32 tools
$
```

The *g* stands for the group and the *o* stands for all other system users; the *-r* denies these users read permission so that they are no longer able to copy the file.

You can also set absolute permissions *r-x--x--x* for the file, i.e. you specify on the command line exactly what permissions you want it to have:

```
$ chmod u=rx,go=x display
$
```

Bear in mind the following points regarding permissions:

- If you have read permission for a file that contains a shell script, you can execute this script using the *sh file* command, even if you do not have execute permission for this file.
- If you have execute permission for a file that contains a shell script, you cannot execute this script unless you also have read permission for this file.

For further information on shell scripts, see chapter 5 *The shell*.

Changing permissions for directories

You can use the *chmod* command to grant or deny permissions for directories as well as files. Simply specify a directory name and not a file name on the command line.

You use the following standard syntax to change the permissions for a directory:

```
chmod who+permission directory_name(s)
chmod who-permission directory_name(s)
chmod who=permission directory_name(s)
```

Bear in mind the effect of changing the permissions for directories.

You require execute permission (*x*) in order to move to a directory or work in a directory. By denying execute permission for a directory, you prevent the user category involved from reading, updating and executing the directory and the files it contains. This overrides any permissions granted for the individual files contained in the directory.

If you grant read permission (*r*) for a directory, the user category involved can only read the contents of the directory using the *ls* command.

Specifying permissions in octal format

There are two ways of entering the *chmod* command.

You can specify the permissions as described above using the lowercase letters *r*, *w* and *x*.

Alternatively, you can specify them using three octal numbers ranging from 0 through 7. The *chmod* entry in the *Commands* manual [5] explains how to specify permissions in this format.

Table 10 summarizes the syntax and function of the *chmod* command.

Command summary			
Command		Important instructions	Arguments
chmod		who +/-/= permission	file directory
Function:		chmod changes the permissions for files and directories.	
Instruction:	who	The user category whose permissions you want to change: u user g group o other users a all three user categories	
	+/-/=	Specifies whether you want to grant or deny permissions: + grant permission - deny permission = grant specified permissions, deny all others	
	permission	Specifies what permissions you want to grant or deny: r read permission w write permission x execute permission	
Arguments:	file	One or more files for which you want to change the permissions.	
	directory	One or more directories for which you want to change the permissions.	
Comments:		You can also specify permissions using an octal number.	
Full syntax:			
chmod [-R]_mode_file_...			
chmod [-R]_mode_directory_...			

Table 10: Summary of the *chmod* command

3.2.9 File types in Reliant UNIX

There are various different file types in Reliant UNIX. Section 3.2 *The structure of the file system* describes files and directories, the two basic types. Reliant UNIX makes a further distinction, however, between the different types of file:

- Files of the type described in section 3.2.1 *Files* are also called "ordinary files".
- A special file represents an input or output device such as a terminal or a hard disk. Chapter 12 *External storage devices* contains detailed information on special files.
- A link is an entry in the file system that enables a file to be addressed under different names. For further information on links, see the *ln* entry in the *Commands* manual [5].
- A named pipe (FIFO) permits interprocess communication and synchronization of processes.

The first character of each line output by the *ls -l* command indicates the file type. For further information, see section 3.3.2 *Listing the contents of a directory: the ls command*.

Because the different file types are not fundamental to an understanding of the file system, they are not covered in more detail in this chapter. For further information on the file types, see the cross-references given above.

3.3 Organizing a directory system

This section introduces you to a number of Reliant UNIX commands that enable you to organize and use a directory structure:

mkdir	enables you to create new directories.
ls	enables you to list the subdirectories and files of a directory.
cat echo	are used in this section for creating files.
rm	enables you to delete files.
rmdir	enables you to delete an empty directory.

3.3.1 Creating a directory: the `mkdir` command

To create a new directory, you use the `mkdir` command (short for **make directory**). You enter the command name, followed by the name of the new directory.

The `mkdir` command has the following standard syntax:

```
mkdir directory_name(s)
```

You can also specify the relative or absolute path name for *directory_name(s)*.

If you want to create a new subdirectory called *project* in the `/home/darkstar` directory, you issue the following command from the `/home/darkstar` directory:

```
$ mkdir project
$
```

You can use the same method to create two more subdirectories called *mail* and *bin* in the `/home/darkstar` directory:

```
$ mkdir mail
$ mkdir bin
$
```

It is also possible to create all three subdirectories (*project*, *mail* and *bin*) at the same time by specifying the three names in a single command line:

```
$ mkdir project mail bin
$
```

Once you have created a directory, you can create any number of subdirectories in it. It is recommended that you create subdirectories according to a logical scheme that will make it easy for you to retrieve information from your files, e.g. by grouping all files belonging to a certain subject area together in one directory. You should also avoid creating an excessive number of directory levels.

You can choose any names for your directories, provided that they conform to the conventions described in section 3.2.1 *Files*.

Table 11 summarizes the syntax and function of the *mkdir* command.

Command summary	
Command	Argument
mkdir	<i>directory</i>
Function:	<i>mkdir</i> creates a new directory.
Argument: <i>directory</i>	Name of the directory you want to create. You can specify more than one directory.
Full syntax: mkdir [<i>-m.mode</i>][<i>-p</i>] <i>directory...</i>	

Table 11: Summary of the *mkdir* command

3.3.2 Listing the contents of a directory: the *ls* command

The *ls* command lists the names of all the files and subdirectories in the directory you specify. If you do not specify a directory name, the command lists the names of all the files and directories in your current directory.

The *ls* command has the following standard syntax:

ls

The following example uses the file system in Fig. 3 to demonstrate how the *ls* command works. You first display the name of the current directory using the *pwd* command. You then list its contents using the *ls* command.

```
$ pwd
/home/darkstar
$ ls
List
bin
john
mail
project
$
```

The command lists the names of the files and directories in alphabetical order, starting with names that have an uppercase letter or a number as their first character.

If you want to list the contents of a different directory, you must specify the path name of the directory involved as an argument of the *ls* command.

ls path_name

You can use either the absolute or the relative path name to specify the desired directory. If, for example, you are working in the */home/darkstar* directory and want to list the contents of the */home/darkstar/project* directory, enter:

```
$ ls project
addresses
schedule
$
```

In this case, *project* is the relative path name from a parent directory (*darkstar*) to a subdirectory (*project*).

You can also use the relative path name to list the contents of a parent directory. For example, the following command line specifies the relative path name from *darkstar* to *home*:

```
$ ls ..
arthur2
darkstar
tsv
$
```

You can obtain the same result by specifying the absolute path name from *root* to the *home* directory:

```
$ ls /home
arthur2
darkstar
tsv
$
```

The *ls* command also enables you to determine whether your current directory contains a specific file. If, for example, you want to find out whether your current directory *project* contains the files *schedule* and *notes*, you enter the *ls* command as follows:

```
$ ls schedule notes
schedule
notes: No such file or directory
$
```

The system displays *schedule* to show you that this file is in the current directory, followed by the message:

```
notes: No such file or directory
```

to indicate that there is no file with the name *notes* in the current directory.

Important *ls* options

You can specify *ls* with certain options in order to obtain specific information about a file or a directory. The options of most interest to you while you are learning about the system are *-a*, *-C*, *-F* and *-l*. Refer to the *ls* entry in the *Commands* manual [5] for details about the other options.

Listing all files in a directory

Some important file names in your home directory start with a dot (e.g. *.profile*). These files are not normally included in the list of files output by the *ls* command. If you want to include them, you must enter *ls* with the *-a* option (short for **all**).

If, for example, you are working in the (*/home/darkstar*) directory and want to obtain a list of all the files contained in it, including the files starting with a dot (*.*), you must enter the command as follows:

```
$ ls -a
.
..
.profile
List
bin
john
mail
project
$
```

Listing contents in short format

If you use *ls* with the *-C* option (short for **columns**), the system lists the files and directories in columns.

If you use *ls* with the *-F* option, the system appends an identifying symbol to certain files. Executable files are identified with an asterisk (*), directories with a slash (/) and links with a commercial at (@).

If you use these two options together, the system lists the contents of your current directory */home/darkstar* as follows:

```
$ ls -CF
List*          bin/          john
mail/          project/
$
```

Listing contents in long format

If you use the `ls` command with the `-l` option (short for long list), you obtain comprehensive information about the contents of a directory (or of a specific file). The long list contains information on

- the permissions
- the number of links
- the owner
- the group name (each user is assigned to a group by the system administrator)
- the size in bytes
- the time of the last modification

If, for example, you enter the `ls -l` command while in the *darkstar* directory, you obtain the following output:

```
$ ls -l
total 30
-rwx----- 2 darkstar phoenix 12301 Nov 02 10:15 List
drwxr-xr-x  3 darkstar phoenix   96 Oct 27 08:16 bin
-rw-----  1 darkstar phoenix   40 Oct 27 10:00 john
drwxr-xr-x  2 darkstar phoenix   80 Nov 08 08:41 mail
drwxr-xr-x  2 darkstar phoenix   64 Nov 01 14:19 project
$
```

The first line of the output (*total 30*) shows the size of the entire directory in 512-byte blocks. Each of the rest of the lines contains information on a directory or file. The first character in each line (*d*, *-*, *l*, *b*, or *c*) indicates the file type as follows:

- d** directory
- ordinary file
- l** link
- b** block special file
- c** character special file
- p** named pipe

Using this key to interpret the example, you can see that the */home/darkstar* directory contains three directories and two ordinary files.

The next nine characters (letters or hyphens) indicate the permissions assigned to the files and directories (for more information on permissions, see section 3.2.8 *Permissions for files and directories*).

The number that appears after the blanks is the link count. For a file, this shows the number of users linked to the file. For a directory, this number is the total number of subdirectories plus two (for the directory itself and its parent directory).

Next, the login name of the owner appears (in this case *darkstar*), followed by the name of the user group to which the file owner belongs (in this case *phoenix*).

The following number shows the size of the file or directory entry in bytes. The date (month, day and time) on which the file or directory was last modified is given next.

The final column contains the name of the file or directory.

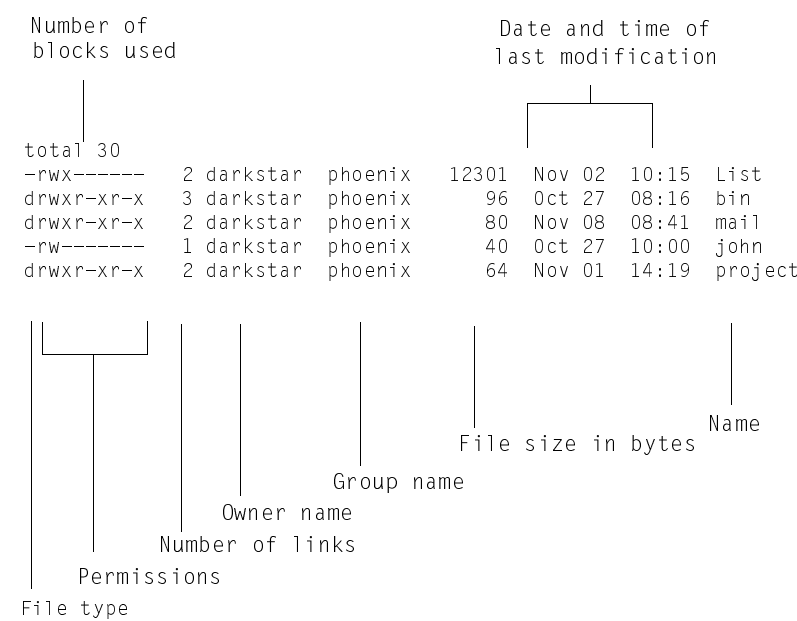


Figure 7: Output produced by the `ls -l` command

Table 12 summarizes the syntax and function of the *ls* command and two of its important options.

Command summary		
Command	Important options	Arguments
ls	-a, -l	<i>file</i> <i>directory</i>
Function:	<i>ls</i> lists the names of files and subdirectories contained in the directories you specify.	
Options:	-a	Lists all entries including names that begin with a dot (.).
	-l	Lists the contents of a directory in long format (including information about the type, permissions, size in bytes and the time of the last modification).
Arguments:	<i>file</i>	One or more files about which you want to obtain information.
	<i>directory</i>	One or more directories about which you want to obtain information.
		If you do not specify an argument, the command lists the contents of the current directory.
Full syntax: ls [_-abcCdfFgiILmnopqrRstux1][_ <i>file</i>] ls [_-abcCdfFgiILmnopqrRstux1][_ <i>directory</i>]		

Table 12: Summary of the *ls* command

3.3.3 Creating files using the cat and echo commands

You normally create files using an editor (see chapters 6 *The display editor vi* and 7 *Other Reliant UNIX editors*). However, you can also use the *echo* and *cat* commands for this purpose. This is particularly useful if you want to redirect the standard output of both these commands to a file. For detailed information on redirecting standard output, see chapter 5 *The shell*.

In order to create files, you must have write permission for the relevant directory. For further information on permissions, see section 3.2.8 *Permissions for files and directories*.

Note:

The *cat* and *echo* commands are used primarily for other purposes. This section describes only a fraction of the functionality of these commands. The *cat* command is described in section 3.4.1 *Displaying the contents of a file: the cat, pg and pr commands*. For a detailed description of the *echo* command, see chapter 5 *The shell*.

In order to create files using *echo* or *cat*, you must redirect the output of these commands using the greater than sign (>). The *echo* and *cat* commands have the following standard syntax when used to create files:

```
echo text > file
```

```
cat > file
```

```
text
```

```
END
```

Caution:

When creating files, remember that a directory may never contain two files with the same name. If a file with the same name already exists in the target directory, the command overwrites it without warning.

When you assign a new file name, make sure that it conforms to the conventions described in section 3.2.1 *Files*.

Suppose that you want to create two files called *jabberwocky* and *verse1* in the */home/darkstar/project* directory. You create *jabberwocky* using the *echo* command; you want this file to contain the character string *A poem by Lewis Carroll*. You use *cat* to create the file called *verse1*; you want this file to contain four lines of verse.

```
$ echo A poem by Lewis Carroll > jabberwocky
```

```
$ cat > verse1
```

```
Twas brillig, and the slithy toves
```

```
Did gyre and gimble in the wabe:
```

```
All mimsy were the borogoves,
```

```
And the mome raths outgrabe.
```

```
END
```

```
$
```

You can now display the contents of these two files on the screen by entering the *cat* command in the following format:

```
cat file(s)
```

You can also append a line to an existing file using the *echo* command. To do so, you must specify the double greater than sign (>>):

```
$ echo Appended line >> jabberwocky
$ cat jabberwocky
A poem by Lewis Carroll
Appended line
```

For more examples of redirecting standard output, see chapter 5 *The shell*. The *cat* and *echo* commands are described in detail in the *Commands* manual [5].

Table 13 summarizes the syntax and function of the *cat* command.

Command summary	
Command	Argument
cat	<i>file</i>
Function:	The <i>cat</i> command writes the contents of all the files specified in the command line to standard output.
Argument: <i>file</i>	One or more files you want to view.
Comments:	By redirecting standard output, you can use <i>cat</i> to create new files or concatenate existing files.
Full syntax: cat [_-suv[et]] [_file]...	

Table 13: Summary of the cat command

Table 14 summarizes the syntax and function of the *echo* command.

Command summary	
Command	Argument
echo	<i>string</i>
Function:	<i>echo</i> writes the command-line arguments to standard output.
Argument: <i>string</i>	Any character string delimited by blanks or tab characters.
Comments:	By redirecting standard output, you can use <i>echo</i> to create new files.
Full syntax: echo [_arg]...	

Table 14: Summary of the echo command

3.3.4 Deleting a file: the `rm` command

You can delete a file using the `rm` command (short for **remove**). This command has the following standard syntax:

```
rm file(s)
```

You can enter several file names, separated by blanks, on the same command line.

Caution:

The system does not make a backup copy of the file you delete. Issuing this command deletes your file permanently. While you are learning about the system, it is therefore advisable to use the `rm` command with the `-i` option (short for **interactive**). When you use this option, Reliant UNIX asks you whether you want to delete the file in each case.

In the following example, you want to delete the files *jabberwocky* and *verse1*. In order to verify whether the `rm` command has been executed successfully, you can use the `ls` command to list the directory contents.

```
$ rm jabberwocky verse1
$ ls
addresses
schedule
$
```

You obtain the same results using `rm -i`, but in this case Reliant UNIX asks you to confirm that you want to delete the file.

```
$ rm -i jabberwocky verse1
jabberwocky: (yes/no) ? y
verse1: (yes/no) ? y
$ ls
addresses
schedule
$
```

The output of the `ls` command shows you that the two files *jabberwocky* and *verse1* have been deleted.

Note:

In order to delete files, you must have write permission for them. For further information on permissions, see section 3.2.8 *Permissions for files and directories*.

Table 15 summarizes the syntax and function of the *rm* command.

Command summary		
Command	Important option	Argument
rm	-i	<i>file</i>
Function:	<i>rm</i> deletes one or more files.	
Option:	-i	Reliant UNIX asks you to confirm that you want to delete each file.
Argument:	<i>file</i>	One or more files that you want to delete.
Full syntax:		
rm _[_ -f] _[_ -i] _[_ -r] <i>file</i> ...		
rm _[_ -r] _[_ -f] _[_ -i] _[_ -r] <i>directory</i> ... _[_ file] ...		

Table 15: Summary of the *rm* command

3.3.5 Deleting a directory: the *rmdir* command

You delete a directory using the *rmdir* command (short for **remove directory**). You can only delete a directory using this command if the directory is empty and you have write permission for it (for further information on permissions, see section 3.2.8 *Permissions for files and directories*).

If you attempt to delete a directory that still contains subdirectories, files or both, the *rmdir* command merely issues the error message:

```
directory not empty
```

The command cannot be executed until all the subdirectories and files have been deleted.

The *rmdir* command has the following standard syntax:

```
rmdir directory_name(s)
```

You can also specify several directory names on the same command line.

The */home/darkstar/tools* directory contains the files *delete* and *cls*. If you attempt to delete the *tools* directory using the *rmdir* command, you merely receive the following error message:

```
rmdir: tools not empty
```

If you want to delete the *tools* directory, you must first delete the two files *delete* and *cls*. You use the *rm* command for this purpose, as described in section 3.3.4 *Deleting a file*.

Once the *tools* directory is empty, you can delete it. Since you cannot delete the directory in which you are currently working, you must move to another directory, e.g. */home/darkstar/bin*, which is the parent directory of *tools*:

```
$ pwd
/home/darkstar/bin
$ rmdir tools
$
```

The *\$* prompt is displayed to indicate that the *tools* directory has been deleted. You can issue the *rmdir* command from any directory that is not a subdirectory of the directory you want to delete. You must specify the absolute or relative path name of the directory you want to delete.

Table 16 summarizes the syntax and function of the *rmdir* command.

Command summary	
Command	Argument
rmdir	<i>directory</i>
Function:	<i>rmdir</i> deletes one or more empty directories.
Argument: <i>directory</i>	One or more directories you want to delete.
Full syntax:	
rmdir [<i>-p</i> [<i>-s</i>]] <i>_directory</i>	

Table 16: Summary of the *rmdir* command

3.4 Accessing files

This section introduces you to some of the Reliant UNIX commands that you need to access and use files in the directory structure. Table 17 summarizes these commands:

Command	Function
cat	displays the contents of a file on the screen.
pg	displays the contents of a file in sections or in pages on the screen.
pr	displays a formatted version of a file on the screen.
cp	copies an existing file.
mv	moves and/or renames a file.
wc	displays the number of lines, words and characters contained in a file.
chown	assigns a new owner to a file or directory.
diff	compares the contents of two files.
grep	searches the contents of a file for a specified character string.
sort	sorts and/or merges the contents of files.

Table 17: Basic commands for using files

The commands *cat*, *pg*, *pr*, *cp*, *mv* and *wc* are basic commands that you need in order to use files. You should familiarize yourself with them before moving on to the other commands.

3.4.1 Displaying the contents of a file: the *cat*, *pg* and *pr* commands

Reliant UNIX provides three commands for displaying the contents of a file: *cat*, *pg* and *pr*.

The *cat* command (short for **concatenate**) displays the contents of one or more files on the screen. You can also use this command to add data to files, concatenate files and, as described in section 3.3 *Organizing a directory system*, create new files.

You use the *pg* command to view the contents of a file one page at a time. This makes the *pg* command particularly useful when you want to read the contents of a lengthy file.

The *pr* command formats the specified files and displays them on your screen. In conjunction with the *lp* command, it redirects the output to a printer (see also the description of the *lp* command in the *Commands* manual [5]).

Displaying the contents of a file: the cat command

The *cat* command displays the contents of one or more files on the screen.

The *cat* command has the following standard syntax:

```
cat file_name(s)
```

Let us assume that you are in the *mail* directory in the sample file system and want to display the contents of the file called *johnson*. Entering the command line as shown obtains the following output:

```
$ cat johnson
                                     February 5, 1992

Mr. Ron Johnson
Layton Printing
52 Hudson Street
New York, N.Y.

Dear Mr. Johnson:

Thank you very much for your inquiry.

.
.
.

If we can be of further assistance to you,
please don't hesitate to call.

We would like to apologize for any inconvenience caused.

Yours truly,

John Howe
$
```

To display the contents of more than one file, type the names of the files on the command line, separated by a blank.

In the following example, the *cat* command reads the files *johnson* and *sanders* consecutively and displays their contents on the screen:

```
$ cat johnson sanders
```

```
February 5, 1992
```

```
Mr. Ron Johnson  
Layton Printing  
52 Hudson Street  
New York, N.Y.
```

```
Dear Mr. Johnson:
```

```
Thank you very much for your inquiry.
```

```
.  
.
```

```
Yours truly,
```

```
John Howe
```

```
February 5, 1992
```

```
Mrs. D.L. Sanders  
43 Nassau Street  
Princeton, N.J.
```

```
Dear Mrs. Sanders:
```

```
Thank you for your offer of a comprehensive  
insurance package.
```

```
.  
.  
.
```

```
Sincerely,
```

```
John Howe  
$
```

If a file contains more lines that can be displayed on the screen, the first lines of the file will roll off the top of your screen. If you want to read these lines, you can suspend output to the screen by pressing the key combination **CTRL** s, and resume it by pressing **CTRL** q.

You can also use *cat* to concatenate the two files to form a single file. To do so, you must redirect standard output, as already described in section 3.3.3 *Creating files using the cat and echo commands*. You use the greater than sign (>) to redirect the output of *cat* to a new file.

Caution:

When creating files, remember that a directory may never contain two files with the same name. If a file with the same name already exists in the target directory, the command overwrites it without warning.

The following example illustrates how to redirect the output of the files *johnson* and *sanders* to a new file called *johnsand*.

```
$ cat johnson sanders > johnsand
$
```

For information on other functions of *cat*, see chapter 5 *The shell*.

Table 18 summarizes the syntax and function of the *cat* command.

Command summary	
Command	Argument
cat	<i>file</i>
Function:	The <i>cat</i> command writes the contents of all the files specified in the command line to standard output.
Argument: <i>file</i>	One or more files you want to view.
Comments:	By redirecting standard output, you can use <i>cat</i> to create new files or concatenate existing files.
Full syntax: cat [<i>_-suv[et]</i>] [<i>file</i>]...	

Table 18: Summary of the *cat* command

Paging through the contents of a file: the *pg* command

The *pg* command displays the contents of one or more files on the screen one page at a time. It is therefore especially useful when you want to read a lengthy file or more than one file.

The *pg* command has the following standard syntax:

```
pg file_name(s)
```

At the bottom of each screen page displayed, you will see a status line containing a colon prompt (:). This indicates that the *pg* command is waiting for you to enter a command.

Table 19 lists some of the *pg* commands you can enter in this status line.

Command	Function
h	Help; lists all the available <i>pg</i> commands
q or Q	Terminate the <i>pg</i> command
	Display the next page of text
-	Display the previous page of text
<i>number</i> 	Display the page of text specified in <i>number</i>
l	Display the next line of text
<i>number</i> l	Display the line of text specified in <i>number</i>
d or ^d	Display an additional half page of text
. or ^l	Redisplay the current page of text
f	Skip the next page of text and display the following one
n	Display the next file specified on the command line
p	Display the previous file specified on the command line
\$	Display the last page of text in the current file
/ <i>pattern</i> /	Search forward in the file for <i>pattern</i>
? <i>pattern</i> ?	Search backward in the file for <i>pattern</i>

Table 19: Some of the *pg* commands

The length of the page displayed depends on the terminal you are using. On a terminal capable of displaying 24 lines, for example, one page comprises 23 lines of text plus the status line at the bottom of the screen containing the colon prompt (:). If the file contains less than 23 lines of text, its page size will be the number of lines in the file plus the status line.

Let us assume that you are in the *mail* directory in the sample file system and you want to view the contents of the *sanders* file one page at a time. Entering the command line as shown obtains the following output:

```
$ pg sanders

                                February 5, 1992

Mrs. D.L. Sanders
43 Nassau Street
Princeton, N.J.

Dear Mrs. Sanders:

Thank you for your offer of a comprehensive
insurance package.
.
.
.
including insurance against damage caused by storms,
:
```

Because the file contains more lines than can be displayed on one screen page, a colon (:) appears at the bottom of the screen to indicate that there are further lines in the file. You can display the next page of text by pressing the  key. This causes *pg* to resume output and display the rest of the file on the screen:

```
the household insurance that
.
.
.
Sincerely,

John Howe
(EOF):
```

Note that the status line at the bottom of the screen contains the character string (*EOF*): (**end of file**). This indicates that you have reached the end of the file.

Once you have finished viewing the file, press . The shell prompt (\$) will appear.

Instead of pressing , you could also enter any *pg* command. For detailed information, see the *pg* entry in the *Commands* manual [5].

For the *pg* command to execute properly, it must know what terminal type you are using. The *pg* command is designed to be flexible enough to run on many different terminals. By specifying the terminal type, you are telling the *pg* command

- how many lines a screen page contains on your terminal
- how many columns a screen page contains

- how to clear the screen
- how to highlight prompts and other characters
- how to delete the current line

The terminal type is specified in the *.profile* file, in which you assign the relevant code to the *TERM* variable (for further information on *TERM* and *.profile*, see chapter 5 *The shell*).

Table 20 summarizes the syntax and function of the *pg* command.

Command summary	
Command	Argument
<i>pg</i>	<i>file</i>
Function:	The <i>pg</i> command displays the contents of one or more files on the screen a page at a time.
Argument: <i>file</i>	One or more files you want to view.
Comments:	After displaying a page of text, the <i>pg</i> command waits for further information, e.g. whether you want to view more text, search for a character string in the text or exit the <i>pg</i> perusal mode. You can also specify a number of options, e.g. to display a section of a file beginning at a particular line or at a line containing a particular pattern.
Full syntax: pg [<i>_-lines</i>][<i>_-p_prompt</i>][<i>_-cefnrs</i>][<i>_=number</i>][<i>_-+lref</i>][<i>_file</i>]...	

Table 20: Summary of the *pg* command

Preparing files for printing: the *pr* command

The *pr* command (short for **print**) formats and outputs the contents of a file. You can output the file contents in paginated form with titles and headers, and with different page lengths and widths.

The *pr* command has the following standard syntax:

pr file(s)

pr prints the text with a default page length of 66 lines, including a header and footer. The header contains five lines: two blank lines, a line containing the date, time and page number, and two additional blank lines. The footer at the bottom of each formatted page consists of five blank lines.

The *pr* command is often used together with the *lp* command to print the contents of a text file on paper (see also the *lp* entry in the *Commands* manual [5]). You can also redirect the output of the *pr* command to another file (for information on redirecting output, see also chapter 5 *The shell*).

Let us assume that you are in the *mail* directory in the sample file system and you want to format and view the contents of the *johnson* file. Entering the command line as shown obtains the following output:

```
$ pr johnson
```

```
Mar  5 15:43 1992 johnson Page 1
```

```
February 5, 1992
```

```
Mr. Ron Johnson  
Layton Printing  
52 Hudson Street  
New York, N.Y.
```

```
Dear Mr. Johnson:
```

```
Thank you very much for your inquiry.
```

```
.  
.   
.
```

```
If we can be of further assistance to you,  
please don't hesitate to call.
```

```
We would like to apologize for any inconvenience caused.
```

```
Yours truly,
```

```
John Howe
```

```
$
```

Because the *johnson* file is 55 lines long and the header is 5 lines long, six blank lines appear at the end of the output. These are the five blank lines of the footer plus an additional blank line, since *pr* extends the output to 66 lines per page.

If you are using a screen that allows you to view 24 lines at a time, the entire 66 lines of the formatted file will be displayed rapidly without a pause. This means that the first 42 lines will roll off the top of your screen. If you want to read these lines, you can suspend output to the screen by pressing the key combination **CTRL** s, and resume it by pressing **CTRL** q.

Note:

By default, the page length is set to 66 lines to fit the American 11" paper size. You can change this default using the `-l` option. `-l72` sets the page length to 72 lines (to fit 12" continuous paper) and `-l70` sets the page length to 70 lines (to fit the A4 paper size).

Table 21 summarizes the syntax and function of the `pr` command.

Command summary		
Command	Important option	Argument
pr	-lpage_length	file
Function:	The <i>pr</i> command formats and displays the contents of one or more files specified in the command line. It outputs the file contents in pages of 66 lines, with a five-line header at the top and five blank lines at the bottom of each page. The header consists of two blank lines, a line containing the date, time, file name and page number, and two additional blank lines.	
Option:	-lpage_length	Number of lines that defines the length of each page output. The default <i>page_length</i> is 66.
Argument:	file	One or more files you want to format and view.
Comments :	The <i>pr</i> command is often used together with the <i>lp</i> command to output one or more text files to a printer.	
Full syntax:		
pr[[[_-columns][_a][_wwidth]][[_-m][_wwidth] [_-e[tab_char][spacing][_i[tab_char][spacing]][[_-dfprt][+page_number] [_-n[separator][char_pos]][[_-ooffset][_lpage_length]][[_-sseparator][_hheader] [_-F][_filei]...		

Table 21: Summary of the `pr` command

3.4.2 Copying a file: the `cp` command

When working with Reliant UNIX, you will often need to copy a file. You might, for example, want to revise a file while leaving the original version intact. The `cp` command (short for **copy**) copies the entire contents of one file into another. The two files involved do not have to be located in the same directory.

The `cp` command has the following standard syntax:

```
cp file copyfile
```

Caution:

When copying files, remember that a directory may never contain two files with the same name. If a file called *filecopy* already exists in the target directory, the command overwrites it with the contents of *file*. While you are still learning about the system, therefore, it is advisable to use the `cp` command with the `-i` option (short for interactive). If you specify this option, Reliant UNIX asks you to confirm that you want to make a copy if the target directory already contains a file called *copyfile*.

Let us assume that you are in the `/home/darkstar/project` directory and want to copy the *schedule* file to a file called *new.schedule*, as shown in the example below. After copying the file, Reliant UNIX displays the prompt. You can use the `ls` command to verify that the new file has been created.

```
$ cp schedule new.schedule
$ ls
addresses
new.schedule
schedule
$
```

If, in the above example, the `/home/darkstar/project` directory already contained a file with the name *new.schedule*, the command would have replaced this file with a copy of *schedule* without issuing a warning. The original version of *new.schedule* would have been deleted. If you enter `cp` with the `-i` option and attempt to copy the *schedule* file to *addresses*, Reliant UNIX asks you for confirmation:

```
$ cp -i schedule addresses
cp: overwrite addresses? n
$
```

If you enter *n*, *schedule* is not copied. The original version of the *addresses* file remains intact. If you enter *y*, the *schedule* file is copied to *addresses*, overwriting the original contents of *addresses*.

Reliant UNIX allows you to have files with the same name, provided that they are located in different directories. You can

- copy a file under the same name to a different directory
- copy a file under a name that exists in the current directory, provided that the target directory does not contain a file with this name

In both cases, the original contents of the file remain intact.

Entering the following command line copies *schedule* from the */home/darkstar/project* directory to the */home/darkstar/mail* directory under the name *addresses*:

```
$ cp -i schedule /home/darkstar/mail/addresses
```

You can also specify a relative path name for *copyfile*:

```
$ cp -i schedule ../mail/addresses
```

In addition, you can copy the *schedule* file to the *home/darkstar/mail* directory under the name *schedule* by entering the following command line:

```
$ cp -i schedule /home/darkstar/mail/schedule
```

In this case, you could also use the relative path name (*../mail/schedule*).

However, it is not always necessary to specify the name of the new file. If you specify a path, but not a name, for the new file on the command line, the *cp* command assigns your new file the name of the original file by default.

Therefore you could also use either of the two command lines shown below:

```
$ cp -i schedule /home/darkstar/mail
```

```
$ cp -i schedule ../mail
```

If you assign a new file name, make sure it conforms to the conventions described in section 3.2.1 *Files*.

Table 22 summarizes the syntax and function of the *cp* command.

Command summary		
Command	Important option	Arguments
cp	-i	<i>file</i> <i>copyfile</i>
Function:	<i>cp</i> copies <i>file</i> to <i>copyfile</i> without overwriting the contents of <i>file</i> .	
Option: -i	If the target directory already contains a file with the name <i>copyfile</i> , Reliant UNIX asks you whether you want <i>copyfile</i> to be overwritten with the contents of <i>file</i> .	
Arguments: <i>file</i>	The file you want to copy. If <i>copyfile</i> is a directory, you can specify several files for <i>file</i> .	
	<i>copyfile</i>	Name of the directory to which you want to copy. If <i>copyfile</i> is a directory, <i>cp</i> copies <i>file</i> to the directory specified in <i>copyfile</i> .
Full syntax:		
cp [<i>-i</i>][<i>-p</i>][<i>-r</i>][<i>-s</i>] <i>file</i> ... <i>copyfile</i>		

Table 22: Summary of the *cp* command

3.4.3 Moving and renaming files: the *mv* command

The *mv* command (short for **move**) moves a file from one directory to another or renames a file or directory. If you move a file to another directory, you can rename it or it can retain its original name.

When you use the *mv* command to rename a file, it has the following standard syntax:

mv file newfile

The *mv* command changes the file name from *file* to *newfile*.

file and *newfile* can be any valid names.

Caution:

When renaming files, remember that a directory may never contain two files with the same name. If a file called *newfile* already exists in the target directory, the command overwrites it with the contents of *file*. While you are still learning about the system, therefore, it is advisable to use the *mv* command with the *-i* option (short for interactive). If you specify this option, Reliant UNIX asks you to confirm that you want to rename the file if the target directory already contains a file called *newfile*.

Let us assume that you want to rename the *addresses* file in the */home/darkstar/project* directory. You give the file the new name *new.addresses* and use the *ls* command to verify that the file has been renamed.

```
$ mv -i addresses new.addresses
$ ls
new.addresses
schedule
$
```

If the target directory already contains a file with the name *new.addresses* and you have set the *-i* option, Reliant UNIX asks you to confirm that you want *new.addresses* to be overwritten.

```
$ mv -i addresses new.addresses
mv: overwrite new.addresses? n
$
```

If you enter *n*, *addresses* is not renamed. The original contents of the *new.addresses* file remain intact. If you enter *y*, the *addresses* file is given the new name *new.addresses* and the original contents of *new.addresses* are overwritten.

You can also specify a directory in place of a file. Note that you may not be working in the directory you want to rename or in any of its subdirectories.

The *mv* command can also be used to move a file from one directory to another. You can either retain the original file name or assign a new name. You use the following standard syntax for moving a file:

```
mv file(s) directory
```

You can specify any valid names, including path names, for files and directories.

The example below illustrates the command line you enter if you want to move the *addresses* file from the */home/darkstar/project* directory to the */home/darkstar/mail* directory. The file retains its original name *addresses*. You can then use the *ls* command to verify that the file has been moved.

```
$ mv -i addresses /home/darkstar/mail/addresses
$ ls
schedule
$ cd ../mail
$ ls
addresses
johnson
sanders
$
```

You can also specify a relative path name in place of the absolute path name. The following command line would thus have the same effect:

```
$ mv -i addresses ../mail/addresses
```

If you want to retain the file name *addresses*, you simply specify the target directory without a file name:

```
$ mv -i addresses /home/darkstar/mail
```

```
$ mv -i addresses ../mail
```

If you assign a new file name, make sure it conforms to the conventions described in section 3.2.1 *Files*.

Table 23 summarizes the syntax and function of the *mv* command.

Command summary		
Command	Important option	Arguments
mv	-i	<i>file</i> <i>newfile</i>
Function:	<i>mv</i> renames a file or moves one or more files to another directory.	
Option: -i	If the target directory already contains a file with the name <i>newfile</i> , Reliant UNIX asks you whether you want <i>newfile</i> to be overwritten with the contents of <i>file</i> .	
Arguments: <i>file</i>	The name of the file you want to move. If <i>newfile</i> is a directory, you can specify several files for <i>file</i> .	
	<i>newfile</i>	Name of the target file to which you want to move or copy <i>file</i> . If <i>newfile</i> is a directory, <i>mv</i> moves <i>file</i> to the directory specified in <i>newfile</i> .
Full syntax:		
mv [<i>_f</i>][<i>_i</i>][<i>_-</i>] <i>_file...newfile</i>		

Table 23: Summary of the mv command

3.4.4 Counting lines, words and characters in a file: the **wc** command

The *wc* command (short for **w**ord **c**ount) reports the number of lines, words and characters in the file(s) specified on the command line. If you specify two or more files, the *wc* command counts the number of lines, words and characters in each individual file and adds them together. In addition, you can specify the *-l*, *-w* or *-c* option on the command line to count the lines, words or characters only.

The *wc* command has the following standard syntax:

wc *file(s)*

The system responds with a line in the following format:

l w c file

where

- *l* represents the number of lines in *file(s)*. *wc* calculates the number of lines by counting how many newline characters there are in the file.
- *w* represents the number of **w**ords in *file(s)*. *wc* considers a word to be any character string separated from another by a blank, a tab character or a newline character.
- *c* represents the number of characters in *file(s)*, including blanks, tabs and newline characters.

For example, to count the lines, words and characters in the *johnson* file in the */home/darkstar/mail* directory, type the following command line:

```
$ wc johnson
60      493      2406 johnson
$
```

The system response tells you that the *johnson* file contains 60 lines, 493 words and 2406 characters.

The following example illustrates how to count the lines, words and characters in more than one file:

```
$ wc johnson sanders
   60      493      2406      johnson
   88      660      3001      sanders
  148     1153      5407      total
$
```

Line, word and character counts for *johnson* and *sanders* are displayed on two separate lines. The sum of each column appears on the last line.

If you want a line, word or character count only, enter the command line in one of the following formats:

`wc -l file` (line count)

`wc -w file` (word count)

`wc -c file` (character count)

If, for example, you enter the command line shown below, you obtain only a line count for the *sanders* file:

```
$ wc -l sanders
    88 sanders
$
```

Similarly, you can use the options *-w* and *-c* to obtain a word count or character count only.

You can use any combination of the *-l*, *-w* and *-c* options.

Table 24 summarizes the syntax and function of the *wc* command.

Command summary		
Command	Options	Argument
wc	-c, -l, -w	file
Function:	wc counts the characters, lines and words in the specified file or files. If more than one file is specified, it totals the individual counts.	
Options:	-c	counts the number of characters
	-l	counts the number of lines
	-w	counts the number of words
Argument:	file	One or more files you want to process.
Full syntax:	wc[_-clw][_file]...	

Table 24: Summary of the *wc* command

3.4.5 Changing the owner of a file: the *chown* command

The *chown* command allows you, the owner of a file, to specify another user as the owner of that file. When you do so, you forfeit all the permissions you had as the original owner of the file.

Note:

Once you have forfeited ownership of a file, only the new owner (or the system administrator) can change the owner using the *chown* command. You are no longer in a position to regain ownership yourself.

The *chown* command has the following standard syntax:

```
chown new_owner file(s)
```

In the following example, the user *darkstar* is the owner of the *display* file (as you can see from the output of the *ls -l* command). This user specifies the user *eddie* as the new owner of the file.

```
$ ls -l display
-r-xr-xr-x  1 darkstar  phoenix    9346 Nov  1  08:06  display
$ chown eddie display
$ ls -l display
-r-xr-xr-x  1 eddie    phoenix    9346 Nov  1  08:06  display
$
```

The system administrator can restrict the use of *chown* to prevent unauthorized users from issuing the command.

Table 25 summarizes the syntax and function of the *chown* command.

Command summary	
Command	Arguments
chown	<i>new_owner</i> <i>file</i>
Function:	The <i>chown</i> command assigns a new owner to a file or directory.
Arguments: <i>file</i>	One or more files or directories to which you want to assign a new owner.
<i>new_owner</i>	Login name of the new owner.
Comments:	Once you have forfeited ownership of a file, only the new user or the system administrator can use <i>chown</i> to assign a new owner.
Full syntax:	
chown [-R][-h] <i>new_owner</i> <i>file</i>	

Table 25: Summary of the *chown* command

3.4.6 Comparing files: the diff command

The *diff* command compares two files and displays the lines in which the files differ. *diff* also tells you how to edit *file1* to make it identical to *file2*.

The *diff* command has the following standard syntax:

```
diff file1 file2
```

The following example compares the files *johnson* and *mcdonough*. *mcdonough* contains the same letter as *johnson*, but with a different address and recipient.

```
$ diff johnson mcdonough
```

```
3,6c3,6
< Mr. Ron Johnson
< Layton Printing
< 52 Hudson Street
< New York, N.Y.
fff
> Mr. J.J. McDonough
> Ubu Press
> 37 Chico Place
> Springfield, N.J.
9c9
< Dear Mr. Johnson:
fff
> Dear Mr. McDonough:
$
```

`3,6c3,6` in the first output line tells you that, if you want *johnson* to match *mcdonough*, you must change lines 3 through 6 in the *johnson* file (3,6...) so that they are identical to the same lines in the other file. The letter *c* stands for change.

As well as *c*, the output may contain the instructions *a* for **a**ppend or *d* for **d**elete.

diff then displays the lines from both files to which these instructions apply, in this case lines 3 through 6 from both files. Lines from *file1* (in this case *johnson*) are prefixed with a less than sign (<) and those from *file2* are prefixed with a greater than sign (>).

Below this, another instruction (9c9) is displayed, followed by the relevant line from each file.

Note that the *diff* command merely identifies the differences between files. It does not change the contents of the two files being compared.

You can use the line editor *ed* to edit the files, however, by specifying the *diff* command with the option *-e*. This creates an *ed* script with which you can change *file1* to match *file2*.

Let us assume that you want to change the *johnson* file to make it identical to the *mcdonough* file. First you create an *ed* script. Then you redirect the output in order to write the script to the *edscript* file, as shown below:


```
$ diff -e johnson mcdonough > edscript
$ cat edscript
9c
Dear Mr. McDonough:
.
3,6c
Mr. J.J. McDonough
Ubu Press
37 Chico Place
Springfield, N.J.
.
$
```

You must then append the *ed* command *w* to the *ed* script in order to save the modified file. You use the *echo* command for this purpose, as shown in the example below.

You then call *ed*. Specify *< edscript* to redirect the input for *ed* so that the editor reads from the script.

The *johnson* file then has exactly the same contents as the *mcdonough* file.

```
$ echo w >> edscript
$ ed johnson < edscript
$ cat johnson
```

```
Mr. J.J. McDonough
Ubu Press
37 Chico Place
Springfield, N.J.
```

```
Dear Mr. McDonough:
.
.
.
$
```

For further information on *ed*, see chapter 7 *Other Reliant UNIX editors*.

Table 26 summarizes the syntax and function of the *diff* command

Command summary		
Command	Important option	Arguments
diff	-e	file1 file2
Function:	The <i>diff</i> command displays the lines in which two files differ and tells you what you must do to make the contents of <i>file1</i> identical to those of <i>file2</i> .	
Option:	-e	<i>diff</i> creates a script for the line editor <i>ed</i> . This script will edit <i>file1</i> so that it matches <i>file2</i> .
Arguments:	file1 file2	The files or directories you want <i>diff</i> to compare.
Comments:	The <i>diff</i> command merely identifies the differences between two files. It does not modify their contents.	
Full syntax:		
diff [<i>-bitw</i>][<i>-D</i> <i>string</i> <i>-C</i> <i>number</i> <i>-c</i> <i>-e</i> <i>-f</i> <i>-h</i> <i>-n</i>] <i>file1 file2</i>		
diff [<i>-bitw</i>][<i>-c</i> <i>-e</i> <i>-f</i> <i>-h</i> <i>-n</i>][<i>-l</i>][<i>-r</i>][<i>-s</i>][<i>-S</i> <i>file</i>] <i>directory1 directory2</i>		

Table 26: Summary of the *diff* command

3.4.7 Searching for a pattern in a file: the *grep* command

You use the *grep* command (short for **g**lobally search for a **r**egular **e**xpression and **p**rint) to search through a file for a section of text that matches a specified pattern. You specify the pattern in the form of regular expressions. A regular expression is a string that can contain both ordinary characters and characters with a special meaning. For the sake of clarity, the following examples use regular expressions containing ordinary characters only. For further information on regular expressions, see the appendix to this guide.

The *grep* command has the following standard syntax:

```
grep pattern file(s)
```

If, for example, you want to search for the word *inconvenience* in the *johnson* file, you enter the following command line:

```
$ grep inconvenience johnson
We would like to apologize for any inconvenience caused.
$
```

The command displays the line in the *johnson* file that contains the specified pattern.

Note that *grep* makes a distinction between uppercase and lowercase letters. For example, it would consider *Inconvenience* and *inconvenience* to be two different patterns.

If the pattern contains more than one word or any character that conveys special meaning to Reliant UNIX (e.g. \$, |, *, ? etc.), you must enclose the entire pattern in single quotes (') (for further information on the special meaning of these and other characters, see chapter 5 *The shell*). If, for example, you want to search for the character string *any inconvenience*, you enter the following command line to obtain the system response shown below:

```
$ grep 'any inconvenience' johnson
We would like to apologize for any inconvenience caused.
$
```

You can also use *grep* to search for a specified pattern in more than one file. For example, if you cannot remember which of the letters *johnson* and *sanders* contains the words *any inconvenience*, enter the following:

```
$ grep 'any inconvenience' johnson sanders
johnson: We would like to apologize for any inconvenience caused.
$
```

The output tells you that the character string *any inconvenience* occurs once in the *johnson* file.

Reliant UNIX provides variations of the *grep* command: *egrep* (processes extended regular expressions) and *fgrep* (processes strings only, but allows you to specify more than one string in each call). For further information, see the entries *grep*, *egrep* and *fgrep* in the *Commands* manual [5].

Table 27 summarizes the syntax and function of the *grep* command.

Command summary	
Command	Important arguments
grep	<i>pattern</i> <i>file</i>
Function:	The <i>grep</i> command searches for a specific pattern in one or more files and displays the lines that contain this pattern.
Arguments: <i>pattern</i>	A string you want <i>grep</i> to search for. <i>grep</i> accepts simple regular expressions for <i>pattern</i> .
<i>file</i>	One or more files in which you want <i>grep</i> to search for a pattern.
Comments:	If the specified character string contains more than one word or any special characters, you must enclose the entire pattern in single quotes '...' on the command line.
Full syntax: grep [<i>-bchilnsvy</i>][<i>-pattern</i>][<i>-file</i>]...	

Table 27: Summary of the *grep* command

3.4.8 **Sorting and merging files: the *sort* command**

The *sort* command sorts the lines in the specified files in the following order:

- Lines beginning with numbers are sorted according to their first number and listed before lines beginning with letters.
- Lines beginning with uppercase letters are listed before lines beginning with lowercase letters.
- Lines beginning with symbols (e.g. *, % or @) are sorted in the order defined by the symbol's ASCII representation.

If you specify more than one file, the command merges the lines before sorting them.

The *sort* command has the following standard syntax:

```
sort file(s)
```

In the following example, the *sort* command merges and sorts the two files *characters1* and *characters2* and displays the results on the screen.

```
$ cat characters1
small
but beautiful
big
BIGGEST
BIGGer
Bigger
**
!!
975
1860
$ cat characters2
!
*
!Exclamation mark
*Asterisk
!exclamation mark
*asterisk
$ sort characters1 characters2
!
!!
!Exclamation mark
!exclamation mark
*
**
*Asterisk
*asterisk
1860
975
BIGGEST
BIGGer
Bigger
big
but beautiful
small
$
```

By redirecting standard output you can create a new file containing the sorted contents of all the specified files.

In addition to combining simple lists as in the above example, the *sort* command allows you to sort lines or parts of lines (known as fields) according to certain criteria. The possible applications are complex, however, and beyond the scope of this section. For a full description of the options available, see *sort* in the *Commands* manual [5].

Table 28 summarizes the syntax and function of the *sort* command.

Command summary	
Command	Important argument
sort	<i>file</i>
Function:	The <i>sort</i> command sorts the lines of a file and displays the results on the screen. If you specify two or more files, the command merges the contents of the files before sorting them.
Argument: <i>file</i>	A file that you want to sort or a number of files that you want to merge and sort.
Comments:	If you do not specify any options on the command line, the command sorts the lines in the order defined by the ASCII representation of the first character of the line.
Full syntax: sort [<i>_-bcdfimMn</i>][<i>_-o</i> <i>output_file</i>][<i>_-r</i>][<i>_-t</i> <i>x</i>][<i>_-u</i>][<i>_-y</i> [<i>kmem</i>]][<i>_-zrecsz</i>] [<i>_-+m</i> [<i>.n</i>][<i>_-m</i> [<i>.n</i>]]][<i>file</i>]...	

Table 28: Summary of the *sort* command

3.5 Summary

Structure of the file system	
Components	
Ordinary file	A collection of characters stored on a hard disk or floppy disk.
Special file	A file used for addressing an input or output device.
Link	An entry in the file system that allows access to a file under different names.
Directory	A collection of files, links and other directories.
Named pipe	Permits interprocess communication and synchronization of processes.
Path names	
Absolute path name	Traces the path from the root directory to the specified file or directory.
Relative path name	Traces the path from the current directory to the specified file or directory.

File names and directory names	
Restrictions	
/	You may not use the slash (/).
? @ # \$ ^ & * \ () [] ' ' ; " < >	You should not use these characters.
+ - .	You should not use these characters as the first character of a name.

Command summary	
Command	Function
cat	displays the contents of all the files you specify on the command line. You can use <i>cat</i> to create new files by redirecting standard output.
cd	changes the current directory.
chmod	changes the permissions for files and directories.
chown	assigns a new owner to a file or directory.
cp	copies a file.
diff	displays the differences between two files.
echo	outputs the command line argument. You can use <i>echo</i> to create new files by redirecting standard output.
grep	searches for a pattern in a file.
ls	lists the contents of a directory.
mkdir	creates a new directory.
mv	moves or renames a file.
pg	displays the contents of a file one page at a time.
pr	formats and outputs the contents of a file.
pwd	displays the path name of the current directory.
rm	deletes a file.
rmdir	deletes a directory.
sort	merges and sorts file contents.
wc	counts the lines, words and characters in a file.

Table 29: Command summary

4 Processes

4.1 Introduction

Whenever you call a Reliant UNIX command or a user-written program, you start a new process. Although there are some exceptions to this, such as the *cd* command, these are not discussed in the following description.

If a number of different users start the same program simultaneously, as is sometimes the case with the *ls* command, separate new processes are started. Each process can spawn new processes, known as child processes. The process that spawns the child is known as the parent process. These processes form a fixed process hierarchy. The uppermost process in the hierarchy is always *sched*, followed by *init*. To view this hierarchy, you can enter the *ps -le* command:

```
$ ps -le □ pg
 F S   UID   PID   PPID  C PRI NI ..... TIME  CMD
39 S    0     0     0    0  0 SY ..... 0:01  sched
10 S    0     1     0    0 39 20 ..... 4:02  init
. :      .      .      .      .      .      .      .
. :      .      .      .      .      .      .      .
10 0    120 17474 17473 14   50 20 ..... 0:00  ps
10 R    120 17473 17458  6   50 20 ..... 0:00  pg
```

Each parent process can spawn more than one child processes. Every process, whether parent or child, has certain identifiers and attributes, which are described in the following sections. Specifically, this chapter deals with

- the process ID (*PID*)
- the user ID (*UID*) and group ID (*GID*) of the process owner
- the process priority
- how to obtain information on active processes
- how to suspend processes
- how to abort active processes
- background processes
- the *nohup* command

The chapter ends with a summary of the most important process management commands.

4.2 Processes and process commands

4.2.1 The process ID (PID)

Each new process is assigned a consecutive process ID (*PID*) that is unique throughout the system. You need to know the process ID if, for example, you want to abort the process using the *kill* command (see section 4.2.6 *Aborting active processes*). As a user, you do not normally know the *PID* under which a process is running. If you start a background process by entering *&* after the command, the relevant process ID is displayed on the screen:

```
$ grep word * > temp &
28223
$
```

If you want to list the processes that are associated with your terminal, enter the *ps* command. This command lists the following information: the process ID (*PID*), the terminal identifier (*TTY*), the cumulative execution time (*TIME*) and the command name (*COMD*).

```
$ ps
  PID TTY          TIME CMD
 3703 pts/3        0:00 ps
 3685 pts/3        0:01 sh
$
```

4.2.2 Real and effective user ID and group ID of a process

The system assigns a process to a user and a group of users by means of the user ID (*UID*) and group ID (*GID*). These IDs also enable the system to check whether a process has permission to access files. A process has two types of UID and GID:

- *effective*
- *real*

If the set-user-ID bit is set for the *owner* of an executable program, the *effective* user ID of the process started when this program is called is the same as the user ID of the *owner* of the program (and not that of the *caller*). In other words, the process runs under the user ID of the program *owner* and can therefore access files for which the *caller* of the program does not have explicit access permission.

The *real* user ID of the process remains that of the program *caller*. If the set-group-ID bit is not set, the *effective* group ID remains the same as the *real* group ID of the process: this is the group ID of the program *caller*.

If the set-group-ID bit is set for an executable program, the *effective* group ID of the process is the group ID of the program *owner*. In other words, the process runs under the group ID of the program *owner*.

The *real* group ID of the process is the group ID of the program *caller*.

The s bits are only useful for executable binary files, such as a compiled C program, not for shell scripts. Although you can use *chmod* to set the s bits for files that contain a shell scripts, the setting will have no effect.

One example of how s bits are used is the *passwd* command, which is stored in the */bin* directory. As a normal user, you are not granted write permission for the password file */etc/passwd*. However, since the set-user-ID bit has been set for the *passwd* command, you can use it to give yourself a new password in the */etc/passwd* file.

4.2.3 Obtaining information on processes

The *ps* command outputs information about processes. If, as shown in the following example, you call the *grep* command as a background process, the process ID is output. Now enter the *ps* command. The system responds by displaying the process ID (*PID*) and the terminal identifier

(*TTY*). This command also tells you the cumulative execution time (*TIME*) and the name of the command executed (*COMMAND*):

```
$ grep word * > temp &
28223
$
$ ps
  PID  TTY    TIME  COMMAND
 28124 tty10  0:00   sh
 28329 tty10  0:04   ps
 28226 tty10  0:04   grep
```

Note that the system outputs a unique process ID for the *grep* command and for all the other active processes. Besides *grep* and *ps*, a further active process is the *sh* command (short for **shell**). This process is active throughout your Reliant UNIX session. The parent process started by the *sh* command is at the top of the hierarchy of processes associated with your terminal. This parent process spawned the two child processes with process IDs 28223 and 28224. The two child processes did not spawn any further processes and are therefore at the lowest level of the hierarchy.

Command summary		
Command	Important options	Important argument
ps	-a, -e, -f, -l -u	 <i>uidlist</i>
Function:	<i>ps</i> displays information about processes.	
Options:	-a	Outputs information about all processes except those not associated with a terminal and process group leaders.
	-e	Outputs information about all processes.
	-f	Outputs a full listing on the processes involved.
	-l	Outputs information on the individual processes.
	-u	Restricts the listing to information about the processes for which the user ID <i>UID</i> or login name of the process owner is specified in <i>uidlist</i> .
Argument:	<i>uidlist</i>	List of user IDs or login names.
Comments:	You need the process ID (<i>PID</i>) output by this command if you want to abort the process using <i>kill</i> .	
Full syntax:		
ps [₋ a ₋ c ₋ d ₋ e ₋ f ₋ l ₋][₋ g ₋ <i>grplist</i>][₋ p ₋ <i>proclist</i>][₋ s ₋ <i>sesslist</i>][₋ t ₋ <i>termlist</i>][₋ u ₋ <i>uidlist</i>]		

Table 30: Summary of the *ps* command

4.2.4 Process priority

Under Reliant UNIX, there are usually several processes competing for a share of CPU time. A control system known as the scheduler determines how CPU time is allocated to processes. The next share of CPU time is normally allocated to the process with the highest priority. To ensure that CPU time is allocated fairly, however, the system changes the priority of each process at regular intervals. When calculating the new priority, the system takes into account the CPU time consumed in the last interval, the size of the process and the length of time a process was preempted, i.e. was waiting for CPU time. There are therefore two types of priority:

- the current priority *PRI*
- the *nice* priority *NI*, defined when the process is started.

The information output by the *ps -l* command includes both the current priority (*PRI*) and the *nice* priority. A high priority number indicates a low priority.

```
$ ps -l
 F S   UID   PID   PPID  C PRI NI   ..... TIME CMD
 10 R   196   3282   3280  2  55 20   .....  0:06 sh
 10 S   196   3261   3259  0  30 20   .....  0:00 sh
 10 0    196   3283   3282 17  55 20   .....  0:00 ps
 10 S   196   3280   3261  0  30 20   .....  0:07 m
$
```

4.2.5 Suspending processes

You can use the *sleep* command to suspend a process, i.e. to delay its execution for a specific length of time. You specify a *time* in seconds in the *sleep* command to define the length of this delay. The *sleep* command has the following standard syntax:

```
sleep time
```

The following example illustrates how to use the *sleep* command from the shell. You start a background process which you want to wake up in ten minutes (600 seconds) and remind you to make a telephone call:

```
$ (sleep 600; echo 'Call Mr. Jones !')&
696
$
```

The process is now running in the background and you can enter further commands at your terminal. After ten minutes, the following message appears on your screen:

Call Mr. Jones !

The *sleep* command is also used frequently within shell scripts. Chapter 5 *The shell* shows you how to write a shell script.

4.2.6 Aborting active processes

You can use the *kill* command to abort a process by sending it a signal. The process that is currently executing either ignores this signal or catches it with a special processing routine. If the process does not catch the signal sent to it, it is terminated when the signal arrives.

The following example illustrates how to abort a process. You use the *kill* command with the signal number 9 and the process ID of the process you want to abort:

```
$ kill -9 28223
28223 Terminated
$
```

Possible signal numbers are:

Number	Symb. name	Meaning
1	SIGHUP	Interrupt the connection with the terminal (hangup)
2	SIGINT	Interrupt via DEL
3	SIGQUIT	Quit
9	SIGKILL	Unconditional process termination (kill)
15	SIGTERM	Software termination

If you enter a nonexistent process ID, you receive the error message:

```
kill: no such process
```

Caution:

When using the *kill* command, you should always make sure that you specify the correct process ID, otherwise you may abort another active process by mistake.

Table 31 summarizes the syntax and function of the *kill* command.

Command Summary	
Command	Argument
kill	<i>process_id</i>
Function:	<i>kill</i> terminates the specified process by means of its <i>process_id</i> (PID).
Full syntax: kill[_-signal]_pid_... kill_-signal_-pgid_... kill_-l	

Table 31: Summary of the *kill* command

4.2.7 Background processes

Some shell commands take a long time to execute. You can use the `&` character to have commands execute as background processes, enabling you to continue working at your terminal during execution. Note that `&` must be the final character on the command line. You use the following standard syntax to execute a command as a background process:

command `&`

Caution:

Interactive shell commands such as *read* cannot execute as background processes since they require input from the user. See section 5.3.3 *Assigning a value to a variable* in chapter 5 *The shell*.

In the following example, the shell executes an extensive search procedure as a background process, with the *grep* command searching for the character string *owl* in the file *laura*.

```
$ grep owl laura &
21940
$
```

When you issue a command as a background process, Reliant UNIX displays the process ID. In the above example, the process ID is *21940*. The prompt on the final line of the above example indicates that you can enter further commands, regardless of whether or not the background process of the *grep* command is still executing. It may be difficult to identify the output of a command executing in the background if a command executed in the foreground is

displaying data on the screen at the same time. You can use the process ID and the *kill* command to abort a background process (for further information, see section 4.2.6 *Aborting active processes*). Once the background process has terminated, the output is displayed on the screen.

If you want a background process to continue executing after you have logged out, you combine the background process with the *nohup* command. For further information, see the next section, 4.2.8 *The nohup command*.

4.2.8 The nohup command

When you log out of Reliant UNIX, all of your processes are aborted. The only exceptions are jobs issued using the *at* and *batch* commands (see *at* and *batch* in the *Commands* manual [5]). If you want a background process to continue executing after you have logged out, you must enter it together with the *nohup* command. The *nohup* command has the following standard syntax:

```
nohup command &
```

You specify *nohup* before the command you want to run as a background process.

Let us assume that you want to use the *grep* command to search all the files in your current directory for the character string *word* and redirect the output to a file called *word.list*. Immediately after entering the command, you want to log out. You enter the following command line:

```
$ nohup grep word * > word.list &  
2918  
$
```


You can abort *nohup* using the *kill* command. Table 32 summarizes the syntax and function of the *nohup* command.

Command summary	
Command	
nohup	
Function:	<i>nohup</i> prevents the execution of a command from being terminated after you have logged out or switched off your terminal. By default, output is written to a file called <i>nohup.out</i> in the current directory or in your home directory.
Full syntax:	nohup <i>_command</i> [<i>_argument</i>]...

Table 32: Summary of the nohup command

4.3 Summary of commands for process control

Command	Meaning
<i>kill</i>	Terminate processes
<i>ps</i>	Output information on processes
<i>nohup</i>	Continue to execute processes after logging out
<i>sleep</i>	Suspend processes
<i>&</i>	Execute processes in the background

Table 33: Summary of commands for process control

5 The shell

5.1 Introduction

The shell is the user interface to the operating system kernel. This means that you, the user, can issue commands to the operating system only via the shell. You must communicate with the shell using a language that it will understand. In section 5.2 *The shell command language*, you will therefore learn about important shell commands and characters with special meaning.

Building on this knowledge, section 5.3 *Shell programming*, which teaches you how to write and execute your own shell programs or scripts.

You can test yourself on what you have learned so far by completing the exercises provided at the end of sections 5.2 *The shell command language* and 5.3 *Shell programming*.

Before starting on this chapter, you should already have worked through the previous chapters, or at least be familiar with the Reliant UNIX file system. You should also have some practical experience of using one of the editors described in this manual.

Note:

This chapter describes the Bourne shell (*sh*). There are other shells under Reliant UNIX, the most important of which are the C shell *csh* and the Korn shell *ksh*. In interactive use, the C shell provides more options than the Bourne shell. In terms of syntax, it is very similar to the C programming language. The Korn shell, like the C shell, is an extension of the Bourne shell. In terms of syntax, however, the Korn shell is to a great extent compatible with the Bourne shell. For more information, see *csh* and *ksh* in *Volume 1* and *sh* in the *Commands* manual [5].

5.2 The shell command language

Chapters 3 *The file system* and 4 *Processes* have already introduced you to two important Reliant UNIX concepts. The following sections describe other Reliant UNIX concepts.

You will learn

- what a command list is and how to enter it
- how to redirect the standard input and standard output of commands
- how to use filters and pipes

You will also become familiar with the most important shell characters with special meaning. A complete list of metacharacters is given under *Shell* in the *Commands* manual [5].

5.2.1 Entering command lists

Reliant UNIX allows you to enter either a single command or a command list on an input line. A command list can consist of:

- several commands, each separated by a command separator
- one or more pipeline or pipelines (to find out what a pipeline is, see section 5.2.3 *Filters and pipes*)

The shell prompt (\$) is normally followed by a blank. You can enter a command or a command list directly after this blank. You terminate your command input by pressing . For further information on entering commands, see section 1.4 *Layout*. If you specify a command list, you must use one of the following characters as a separator:

Separator in command list	Meaning
;	causes the individual commands or pipelines in a command list to be executed <i>sequentially</i> .
&	causes the individual commands or pipelines in a command list to execute as background processes.
&&	reads the exit status of a command or pipeline. A command that follows && is executed only if the previous command or pipeline executed successfully (exit status 0).
	reads the exit status of a command or pipeline. A command that follows is executed only if the previous command or pipeline did <i>not</i> execute successfully (exit status not 0).

Now try entering the following examples of command lists:

Example 1:

Go to your home directory and display the current directory. Then go to the subdirectory *\$HOME/bin* and display the current directory:

```
$ cd;pwd;cd $HOME/bin;pwd
/usr2/guide
/usr2/guide/bin
$
```

Example 2:

Enter *pwd* to display the current directory. Once this command has executed successfully, you want *ls* to display the contents of the current directory:

```
$ pwd && ls
/usr2/guide
afile
bfile
cfile
$
```

Example 3:

Display the path names and files with the name *tmp* that are located in the current directory or in any of its subdirectories. You only want the subsequent *ls* command to be executed if the previous command executed successfully. Enter the following:

```
$ find . -name tmp -print && ls
afile
bfile
cfile
$
```

You may at first be confused by this result: the *ls* command was executed even though the previous *find* command did not find any files called *tmp*. Only the contents of the current directory were output. However, this output is correct; *find* was in fact executed successfully. It simply did not find any files called *tmp*, since there is no file with this name in the current directory or in any of its subdirectories. The first command therefore returned the exit status 0 and *ls* was executed as a result.

5.2.2 Redirecting input and output

Under Reliant UNIX, many commands read from standard input (the keyboard) and write to standard output (the screen). Standard error is also written to the screen. You can change this default assignment. For example, you can tell a command to read its input from a file instead of from standard input, or write its output to a file instead of to standard output or standard error.

If you want to redirect standard error to a file or a printer, you must specify the file descriptor 2. When redirecting standard input or standard output, however, you can omit the file descriptor. If you specify a file descriptor, the standard input or standard output is redirected to the file associated with this file descriptor. You can specify the following file descriptors:

File descriptor	Associated standard input/ standard output
0	Standard input
1	Standard output
2	Standard error

This section describes the following operators that are used to redirect input and output:

- the less than character (<)
- the greater than character (>)
- the double greater than character (>>)

Redirecting standard input from a file: the less than character (<)

You use the less than character (<) to redirect standard input. The command then reads its input data from a file instead of standard input. The standard syntax is:

Alternative 1: *command < file*

Alternative 2: *command 0< file*

Suppose that you want to use the *mail* command to send a message to another user with the login name *colleague* (see chapter 9 *Electronic mail*). The message is already stored in the *report* file.

To save entering the message text again, you can tell the command to take its input from this file:

```
$ mail colleague < report
$
```

Redirecting standard output to a file: the greater than character (>)

You use the greater than character (>) to redirect standard output. The command then writes its output to a file instead of displaying it on the terminal. The standard syntax is:

Alternative 1: *command* > *file*

Alternative 2: *command 1*> *file*

If you redirect the output of the command to a file that does not yet exist, a new file is created with the name specified. First check whether your current directory contains a file called *junk* by entering the *ls* command with the file name:

```
$ ls junk
junk: no such file or directory
$
```

There is no file called *junk*. The next example illustrates how to redirect the output of *ls* to the *junk* file. If *junk* does not yet exist, the command creates a new file with this name. You can then use the *cat* command to view the contents of *junk*:

```
$ ls > junk
$ cat junk
afile
bfile
cfile
$
```

It is particularly useful to redirect output to a file in cases where the output of a command is too long to fit onto a single screen page, e.g. when you use the *spell* command. *spell* compares every word in a file with an internal spelling list and outputs any words that do not match the list. *spell* also reports words not found in the list, such as proper names, as spelling mistakes. If the list of misspelled words does not fit on one screen page, it is displayed in scrolling mode and disappears before you can read it. You can circumvent this problem by redirecting the output of *spell* to a file. In the following example, *spell* checks a file called *memo* and writes a list of misspelled words to a file called *misspell*:

```
$ spell memo > misspell
```

Table 34 summarizes the syntax and function of the *spell* command.

Command summary	
Command	Important argument
spell	<i>file</i>
Function:	<i>spell</i> checks the spelling of words in a file against a spelling list. Words that are not present in the list are displayed on the terminal.
Argument: <i>file</i>	File you want to check for spelling errors.
Comments:	You can redirect the list of misspelled words to a file.
Full syntax: spell [<i>-b</i>][<i>-l</i>][<i>-v</i>][<i>-x</i>][<i>+local_file</i>][<i>file</i>]...	

Table 34: Summary of the *spell* command

Caution:

Before using `>` to redirect the output of a command to a file, you should make sure that a file with the specified name does not already exist. Otherwise, the command overwrites the contents of the existing file without warning you first. You can avoid overwriting this file, however, by using the double greater than character (`>>`) instead of the single greater than character (`>`). If you specify `>>`, the output is appended to the existing file, as explained in the next section.

**Appending output to an existing file:
the double greater than character (>>)**

You specify the double greater than character to append the output of a command to a file. If the output file does not yet exist, it is created when you use `>>`. The standard syntax is:

command `>>` *file*

The following example illustrates how to append the output of the *cat* command to an existing file. The *cat* command writes the contents of the files you specify as arguments to standard output. You first execute the *cat* command for each file without redirecting the output. You then append the contents of the *result* file to the *calculate* file by executing the *cat* command for *result* and redirecting the output to *calculate*:


```
$ cat calculate
two plus two
equals
$ cat result
four
$ cat result >> calculate
$ cat calculate
two plus two
equals
four
$
```

Redirecting standard error for a command

You specify `2>` to redirect standard error. The filter `2` in front of the greater than character indicates that you want to redirect standard error, not standard output. The standard syntax is:

command 2> file

In the next example, you specify an illegal option for the `ls` command and want error output to be written to the `trash` file instead of to the screen:

```
$ ls -y 2> trash
$ cat trash
ls: illegal option --y
usage: ls -lRadCxmnlogrtpucpFbqisfL [files]
$
```

5.2.3 Filters and pipes

Filters are programs or commands that read from standard input and write the result to standard output. Many of the Reliant UNIX commands are implemented as filters. You can also use the pipe mechanism to join filters together. Pipes are powerful tools that allow you to take the output of one command and use it as input for another command without creating temporary files. The `|` character is the symbol for a pipe. A command line that contains several pipes joined together using this symbol is called a pipeline. The standard syntax is:

command1 | command2...

The command to the left of the pipe symbol creates an output that is passed to the command to the right of the pipe symbol as its input. The output of *command1* is therefore used as input for *command2*, and the output of *command2* could in turn be used as input for *command3*.

Let us assume that you want to know how many entries a directory contains. You could do this in several stages: first write the output of the *ls* command to a file called *temp*, then count the number of lines in *temp* using *wc -l* and finally delete the file:

```
$ ls > temp
$ wc -l < temp
    41
$ rm temp
$
```

You can obtain the same result much more quickly and easily by joining these commands together using pipes. This also saves having to create and delete a temporary file:

```
$ ls | wc -l
    41
$
```

Further examples of pipe usage

Example 1:

Search two files for all customers who live in *minneapolis* and display these lines in alphabetical order. The files have the following structure:

Contents of the file *customers1*:

```
luckman    minneapolis  customer no.:160349
arctor     minneapolis  customer no.:300350
hawthorne  minneapolis  customer no.:190495
freck      detroit      customer no.:310846
tinbane    detroit      customer no.:301462
```

Contents of the file *customers2*:

```
hamilton   minneapolis  customer no.:160851
trask      minneapolis  customer no.:300461
lee        minneapolis  customer no.:190851
steinbeck  detroit      customer no.:311249
bacon      detroit      customer no.:302368
```

First read the two files *customers1* and *customers2* using the *cat* command. Specify the pipe symbol (*|*) to pass the files to the *grep* command. *grep* then passes all lines that contain the specified search key to the *sort* command, which outputs them in alphabetical order:

```
$ cat customers1 customers2 | grep minneapolis | sort
arctor      minneapolis  customer no.:300350
hamilton    minneapolis  customer no.:160851
hawthorne   minneapolis  customer no.:190495
lee         minneapolis  customer no.:190851
luckman     minneapolis  customer no.:160349
trask       minneapolis  customer no.:300461
```

Example 2:

Extract the time from the *date* command. First execute the *date* command to display the current day, date and time:

```
$ date
Sat Dec 24 13:12:32 1988
$
```

The time is displayed in columns 12 through 19. If you want to obtain the time only (without the date), you can use the output of the *date* command as the input for the *cut* command. *cut -c12-19* extracts columns 12-19 from a line. The command line and resulting output are shown below:

```
$ date | cut -c12-19
13:14:56
$
```

Using the output of a command as an argument

You can capture the output of most commands and use it as an argument in a command line. You do this by enclosing the command in backquotes ('...') and placing it on the command line in the position where you want its output to be interpreted as an argument. This procedure is known as command substitution.

Let us assume that you want to substitute the output of the *date* and *cut* pipeline used in the last section for the argument in the *banner* command. Enter the following command line:

```
$ banner `date | cut -c12-19`
```

The system prints a banner displaying the current time. Further information on *banner*, *date* and *cut* is given in the *Commands* manual [5] under the relevant entries.

5.2.4 Shell metacharacters

In section 5.2 *The shell command language* and its first three subsections, you were introduced to some of the characters with special meaning used when entering commands, redirecting input and output, and using filters and pipes. These characters are also listed in the summary at the end of this chapter, together with all the other characters covered in the following sections.

This section describes the special characters you can use to

- replace arguments with matching file names (file name generation)
- escape other special characters

File name generation

The following sections describe the metacharacters `*` (asterisk), `?` (question mark) and `[ ]` (square brackets).

You can specify these characters in patterns that you use to search for file names or partial file names. This is an easy way of specifying files or groups of files as command arguments. For example, you might use these metacharacters to refer to all file names that contain the letter *a* or that consist of five characters.

The metacharacter that stands for any string: the asterisk ()*

The `*` character stands for any string, including an empty string. You can use it to specify a full or partial file name. If you specify `*` on its own, the shell replaces it with all the file and directory names in the current directory that do not start with a dot (`.`). To see the effect of the `*` character, specify it as an argument of the `echo` command:

```
$ echo *
```

The `echo` command displays its arguments on the screen. The `*` passes all the file names in the current directory to the `echo` command as arguments. The command then outputs these file names.

Caution:

The asterisk (`*`) specified on its own stands for all file and directory names. If, for example, you enter `rm *`, you delete all the files in your current directory. For this reason, you should use the asterisk with care.

Let us assume that you have created a number of files and named them *report*, *report1a*, *report1b.01* and *report25*. To find out how many files you have created, you can use the *ls* command with the *** metacharacter to list all the file names that begin with the string *report*:

```
$ ls report*
report
report1a
report1b.01
report25
$
```

*** stands for any number of characters after the string *report*, including no characters at all. Note that the asterisk returns the file names in numerical and alphabetical order.

Another example: Your current directory contains the files *Andrew_Adams*, *Paul_Lang*, *Jane_Peters*, *Fran_Smith* and *Antony_Myers*. Choose a character that the file names in your directory have in common, e.g. the lowercase *a*. You can then tell the *ls* command to list all file names that contain this character:

```
$ ls *a*
Andrew_Adams
Fran_Smith
Jane_Peters
Paul_Lang
```

The resulting list contains all the file names in your current directory except *Antony_Myers*. The system outputs all the file names in your current directory that contain a lowercase *a*.

*** can stand for characters in any part of the file name. If, for example, you know that different file names have the same first and last letter, you can list them on this basis. Suppose, for example, that your directory contains the files *FATE*, *FE*, *FADED_LINE* and *F123E*. You can specify the asterisk as shown below to list all the file names that start *F* and end with *E*:

```
$ ls F*E
```

The system outputs a list of file names, sorted in the following order:

```
F123E
FADED_LINE
FATE
FE
```

The list of file names is sorted in ascending order; numbers are listed first, followed by uppercase letters and finally lowercase letters.

`*` provides you with even more extensive options. For example, you can specify it with the `ls` command in order to find all the files called *memo* that are located either in the current directory or in any of its subdirectories.

```
$ ls */memo
CORR/memo
GUIDE/memo
```

The metacharacter that stands for any single character: the question mark (?)

The question mark (`?`) stands for any single character of a file or directory name except a leading dot. Suppose that you have written twelve chapters of a book and you now want a list of chapters one through nine. Your directory contains the files *chapter1* through *chapter12*. You can specify the `?` in the `ls` command in order to list all the chapter names that begin with the string *chapter* and end with any single character:

```
$ ls chapter?
chapter1
chapter2
chapter3
chapter4
chapter5
chapter6
chapter7
chapter8
chapter9
$
```

The system lists all the file names that match this pattern. The question mark stands for any single character and may occur more than once in a file name. The following command lists the files *chapter10*, *chapter11* and *chapter12* on the screen, but not *chapter1* through *chapter9*.

```
$ ls chapter??
```

If you want to list all the chapters in the current directory, simply use the asterisk (`*`):

```
$ ls chapter*
```

Metacharacters that stand for a set of characters: square brackets []

You use the square brackets (`[ ]`) if you want the shell to search for any one of several possible characters that may appear in one position in the file name. Let us assume that your directory contains the four files *pat*, *fat*, *mat* and *rat*. If you include `[prf]` in your search pattern, the shell will look for file names that contain the letter *p*, *r* or *f* in the specified position:

```
$ ls [prf]at
fat
pat
rat
$
```

The command displays all file names that start with the letter *p*, *r* or *f* and end in the letters *at*. A set of characters specified in brackets in this way is known as a *character class*.

You can also use square brackets to specify a range of characters, whether numbers or letters. Let us assume that your directory contains the files *chapter1*, *chapter2*, *chapter3*, *chapter4*, *chapter5* and *chapter6*. Enter the following command:

```
$ ls chapter[1-5]
```

In this case, the shell looks for files with the file name *chapter1* through *chapter5*, i.e. all file names that end with a number in the range [1-5].

A character class may also specify a range of letters. If you specify *[A-Z]*, the shell will look for uppercase letters only; if you specify *[a-z]*, it will look for lowercase letters only.

Table 35 summarizes the functions of these metacharacters.

Character	Function
*	Stands for any string, including an empty string, but not file or directory names that begin with a dot.
?	Stands for any single character except a leading dot.
[]	Stands for any one of the characters enclosed in the square brackets.
[-]	Stands for any one of the characters in the range specified in the square brackets.

Table 35: Summary of the metacharacters for file name generation

Escaping shell metacharacters

In addition to the metacharacters already discussed, the shell recognizes several other useful characters with special meaning. This section describes the backslash (**), single quote (*'*) and double quote (*"*).

Escaping metacharacters using the backslash (\)

The shell interprets the backslash (\) as an escape character that allows you to cancel the special meaning of the character immediately after it. The following exercise illustrates how the backslash works. Use an editor with which you are familiar, e.g. *vi*, to create a file called *comet* containing the following three lines:

```
The * in the first line
is not in the second line.
Why not put it in the third line???
```

Use the *grep* command to search for the asterisk (*) in the *comet* file:

```
$ grep \* comet
The * in the first line
$
```

The *grep* command searches through the text to find the asterisk and displays the line in which it occurs. If you did not specify a backslash, the asterisk would be interpreted as a metacharacter.

Escaping metacharacters using single quotes (') and double quotes (")

Another way of escaping metacharacters is to use single quotes (') and double quotes ("). Single quotes ('...') escape all metacharacters except single quotes. Similarly, double quotes ("...") escape all metacharacters except double quotes, the dollar sign (\$), the backslash (\) and the backquote (`). An advantage of using single and double quotes is that they allow you to specify several metacharacters at once. This results in a more concise and legible format than if you were to escape each metacharacter individually using the backslash. Let us assume that you want to search for the following line in the *comet* file:

```
Why not put it in the third line???
```

Escape the three question marks by enclosing them in single quotes:

```
$ grep '???' comet
Why not put it in the third line???
```

Escaping blanks using single quotes ('...') and double quotes ("...")

Single and double quotes, like the backslash (\), are used as escape characters for canceling the special meaning of blanks. The shell interprets a blank on a command line as a delimiter between the individual command arguments. Both single and double quotes allow you to cancel this special meaning.

If, for example, you want to locate two or more words that appear together in a text, you must pass these words to the *grep* command as a single argument by enclosing them in single or double quotes. If you want to find the string *first line* in your *comet* file, you must enter the following command line:

```
$ grep 'first line' comet
The * in the first line
$
```

grep searches for the string *first line* and displays the line that contains it on the screen. If you had not enclosed the string in single (or double) quotes, the word *line* would have been interpreted as an input file along with *comet*. *grep* would have searched for the string *first* in a file called *line*. If there had been no file called *line*, you would have received the following error message:

```
grep: can't open line
```

grep would then have searched for the string *first* in the *comet* file and displayed the line containing this string on the screen.

It is useful to be able to cancel the meaning of the blank as a delimiter when using certain commands, such as *banner*. This command displays a message across the screen in large letters. You enter the *banner* command with a message that consists of an argument or several arguments (normally words) separated by blanks. The *banner* command interprets these blanks as delimiters and displays each argument on a separate line. If you want to display more than one argument on the same line, you must enclose the words you want on one line in quotes. If, for example, you want to display a birthday greeting over four lines, you enter the following command:

```
$ banner happy birthday to you
```

The following command displays the message over three lines:

```
$ banner happy birthday "to you"
```

Note that the words *to* and *you* will now appear on the same line. The blank between them has lost its meaning as a delimiter.

Table 36 summarizes the special characters covered in this chapter:

Character	Function
* ? [] [-]	The asterisk, question mark and square brackets provide a shortcut for specifying file names using search patterns.
&	The ampersand places commands in background mode, leaving your terminal free for other tasks.
;	The semicolon separates commands specified on the same command line.
\	The backslash escapes special characters such as *,?, [], &, ;, >, < and .
' ...'	Single quotes cancel the meaning of the blank as a delimiter and escape all special characters.
' ...'	Used in pairs, backquotes substitute the output of a command for the argument in a command line.
"..."	Double quotes cancel the meaning of the blank as a delimiter and escape all special characters except \$ and '.
>	The greater than character redirects the output of a command to a file (overwriting the original contents of that file).
<	The less than character redirects input so that a command reads its input from a file.
> >	The double greater than character appends the output of a command to an existing file.
	The pipe symbol causes the output of one command (to the left of the symbol) to be used as input for the next command (to the right of the symbol).
\$	The dollar sign is used in positional parameters and user-defined variables (see section 5.3.2 <i>Variables</i>). The dollar sign is also the default shell prompt.

Table 36: Summary of the shell special characters

5.3 Shell programming

You can use the shell to create programs or new commands. These programs are called *shell programs* or *shell scripts*. In this section, you will learn how to create and execute shell scripts with commands, variables, positional parameters, return codes and basic programming control structures. In order to create shell scripts, you must be familiar with an editor (as described in 6 *The display editor vi*, for example).

5.3.1 Shell scripts

Creating a simple shell script

Let us begin by creating a simple shell script that will perform the following tasks:

- display the name of the current directory
- list the contents of this directory
- display the following message on your terminal:

```
This is the end of the shell script.
```

Use an editor to create a file called *dl* (short for **d**irectory **l**ist) and enter the following:

```
pwd
ls
echo This is the end of the shell script.
```

Now save these lines and quit the editor. You have just written a shell script. You can use the *cat* command to display the contents of this file. Your screen will look like this:

```
$ cat dl
pwd
ls
echo This is the end of the shell script.
$
```

Executing a shell script

You will now learn about two different ways of executing a shell script.

The first way is to call the new command *dl* by entering *sh*; this is the easiest way to execute a shell script. The path name of the current directory is displayed first, followed by a list of the files in the current directory and finally the comment *This is the end of the shell script*. Now try entering *sh* yourself:

```
$ sh dl
/usr2/tom
afile
bfile
cfile
This is the end of the shell script.
```

Alternatively, you can execute your script without entering *sh*. However, you must first make *dl* an executable file using the *chmod* command (see *chmod* in the *Commands* manual [5]). You can use the *chmod* command to grant execute permission to the user, i.e. yourself. To do this, you specify *u* (for **u**ser) and *+x* (for **e**xecute):

```
$ chmod u+x dl
```

dl is now an executable file. You can then use the *ls -l* command to verify that the user has been granted execute permission. For a description of the *ls -l* command, see the *Commands* manual [5]. Now enter the command:

```
$ ls -l
total 2
-rw----- 1      name    name    3661   Nov  2  10:28 mbox
-rwx----- 1      name    name      48   Nov 15  10:50 dl
$
```

You can see that the *dl* file now has execute permission (*x*). Execute the shell script:

```
$ dl
```

You obtain the same output as when you entered *sh dl*.

Note:

A third way of executing a shell script is to use the (dot) command (*.*). This is described in section 5.4 *Modifying your login environment*. Unlike *sh*, the dot command starts the login shell and not a subshell. See also *sh* in the *Commands* manual [5].

Creating a bin directory for executable files

To make your shell scripts accessible from all your directories, you can create a directory called *bin* under your home directory and move all your shell files to there. You must also set your environment variable *PATH* to include the *bin* directory:

```
PATH=$PATH:$HOME/bin
```

For further information on the *PATH* variable, see the description of *Shell variables* in section 5.4 *Modifying your login environment*.

Let us assume that you want to create *bin* as a subdirectory of your home directory and copy the *dl* file from *home* to *bin*. First enter the *cd* command to make sure that you are in your home directory. Then use *mkdir* to create the *bin* subdirectory. Finally, use *mv* to move the *dl* file to the *bin* subdirectory.

```
$ cd
$ mkdir bin
$ mv dl bin/dl
```

Now go to the *bin* directory using the *cd* command:

```
$ cd bin
```

Check whether the *dl* file is still executable:

```
$ ls -l
```

The output should show that *dl* is still executable.

Table 37 summarizes your new shell script *dl*.

Summary	
Shell script <i>dl</i> - display the directory path and list the directory content (user-defined)	
Command	Arguments
dl	<i>none</i>
Function:	<i>dl</i> displays the name and contents of the current directory.

Table 37: Summary of the *dl* shell script

You can give the *bin* directory a different name; if you do so, you must change your *PATH* environment variable accordingly.

Caution:

You can give your shell script almost any name, provided that it is not identical to a Reliant UNIX command name. If your shell script has the same name as a system command, the system will execute the system command in preference to your shell script. The reason for this is that the system searches for commands in the directory that is defined first in the *PATH* variable. Your own current directory, represented by a dot, appears last in the path specification and is therefore the last directory to be searched. The easiest way to avoid problems of this kind is to give your shell script a name that is not identical to the name of a Reliant UNIX command. For example, you should not give your script a name such as *ls* or *mv*.

5.3.2 Variables

Variables, like files, are basic data objects manipulated in shell scripts. This section describes three variable types and explains how to use them:

- positional parameters
- special parameters
- keyword parameters

Positional parameters

A positional parameter is a variable within a shell script. It takes its value from an argument that you specify on the command line when invoking the script. Positional parameters consist of numbers with a preceding dollar sign (\$) (*\$1*, *\$2*, *\$3* etc.).

A shell script may reference up to nine positional parameters. Enter the shell program *shell.prog* as shown below:

```
$ shell.prog pp1 pp2 pp3 pp4 pp5 pp6 pp7 pp8 pp9
```

When this shell script is called, the positional parameter *\$1* within the program is assigned the value *pp1*, positional parameter *\$2* is assigned the value *pp2*, and so on.

To practice positional parameter substitution, create a file with the name *pp* (short for **p**ositional **p**arameters). Then enter the *echo* commands shown below. Enter the command lines so that, when you execute the *cat* command for your completed file, you will obtain the following output:

```
$ cat pp
echo The first positional parameter is: $1
echo The second positional parameter is: $2
echo The third positional parameter is: $3
echo The fourth positional parameter is: $4
$
```

Now use the *chmod* command to make the shell script *pp* an executable file. Then invoke the shell script with the arguments *one*, *two*, *three* and *four*:

```
$ chmod u+x pp
$ pp one two three four
The first positional parameter is: one
The second positional parameter is: two
The third positional parameter is: three
The fourth positional parameter is: four
$
```

The arguments you specified when calling the shell script have replaced the positional parameters *\$1* through *\$4*. The next command line shows the shell script *birthday*, which mails a greeting to the user with the specified login name:

```
$ cat birthday
banner happy birthday | mail $1
$
```

Now try sending yourself a birthday greeting. If your login name is *sue*, your command line will look like this:

```
$ birthday sue
```

Table 38 summarizes the syntax and function of the shell script *birthday*.

Summary	
Shell script <i>birthday</i> - mail a banner birthday greeting	
Command	Argument
<i>birthday</i>	<i>login_name</i>
Function:	<i>birthday</i> mails the message 'happy birthday' in large-size letters to the user specified.

Table 38: Summary of the birthday shell script

The *who* command lists all users who are currently logged in. How can you create a simple shell script that will tell you if a user with a particular login name is currently working on the system?

Enter the following line in a file called *whoson*:

```
who | grep $1
```

The *who* command lists all users currently logged in; the *grep* command searches this output for a line containing the string you specify for the positional parameter *\$1*. Check the contents of the *whoson* shell script again:

```
$ cat whoson
who | grep $1
$
```

Once you have made the file executable using *chmod u+x*, call the *whoson* script, specifying the login name *sue* as the argument, and look at the output:

```
$ whoson sue
sue  tty26          Jan 24 13:35
$
```

The shell script has substituted for the login name *sue* the parameter *\$1*.

If the user *sue* is not currently logged in, the *grep* command cannot be executed and you receive no output from *whoson*.

Table 39 summarizes the syntax and function of the shell script *whoson*.

Summary	
Shell script whoson - display login information if user is currently logged in (user-defined)	
Command	Argument
whoson	<i>login_name</i>
Function:	If the user is logged in, <i>whoson</i> displays the login name of the user, the terminal ID and the date and time the user logged in.

Table 39: Summary of the whoson shell script

The shell command line can contain up to 128 arguments; however, a shell script can only reference up to nine positional parameters (*\$1* through *\$9*) simultaneously. You can circumvent this restriction using the *shift* command, described under *shift* in the *Commands* manual [5]. You can use the special parameter *\$**, described in the next section, to reference the values of all arguments specified on the command line.

The special parameters *\$#* and *\$**

When used in a shell script, the *\$#* parameter described below contains the number of arguments with which the shell script was called. Its value can be used anywhere in the shell script.

Create a shell script called *get.num* that outputs the number of arguments with which it is called.

```
$ cat get.num
echo The number of arguments is: $#
$
```

If you call the script with the four parameters shown below, you will obtain the result shown (provided that you did not forget to make *get.num* an executable file):

```
$ get.num Test out this program
The number of arguments is: 4
$
```

Table 40 summarizes the shell script *get.num*.

Summary	
Shell script <i>get.num</i> - count and display the number of arguments (user-defined)	
Command	Argument
<i>get.num</i>	<i>string</i>
Function:	<i>get.num</i> counts the number of arguments specified in the command line and displays the total.
Note:	This command demonstrates the special parameter <i>\$#</i> .

Table 40: Summary of the *get.num* shell script

When referenced in a shell script, the parameter *\$** described below contains a string consisting of all the arguments with which the shell script was called. This parameter, unlike the positional parameters *\$1* through *\$9*, can contain virtually any number of arguments.

The next example uses a simple shell script to demonstrate how *\$** works. Create a shell script called *show.param* that uses the *echo* command to display all parameters. Enter the following line in the *show.param* file:

```
echo The parameters for this command are: $*
```

The shell script *show.param* echoes all the arguments with which the command is called. Make *show.param* an executable file and test it using the following parameters:

```
Hello. How are you?
```

Now call the shell script with these parameters and see what output you receive:

```
$ show.param Hello. How are you?  
The parameters for this command are: Hello. How are you?  
$
```

Now try calling *show.param* with more than nine arguments:

```
$ show.param one two 3 4 5 six 7 8 9 10 11  
The parameters for this command are: one two 3 4 5 six 7 8 9 10 11  
$
```

Again, *show.param* displays all the arguments specified in the command line. The *\$** parameter is particularly useful if the arguments of a shell command contain metacharacters for file name expansion.

Try using file name expansion with the shell script *show.param*. Suppose that your directory contains several files, each containing one chapter of a book. The files are called *chap1*, *chap2* ... *chapF*. *show.param* displays a list of all these files.

```
$ show.param chap?  
The parameters for this command are: chap1 chap2 chap3  
chap4 chap5 chapB chapF  
$
```

Table 41 summarizes the shell script *show.param*.

Summary	
Shell script <i>show.param</i> - display all positional parameters (user-defined)	
Command	Arguments
<i>show.param</i>	Any positional parameters
Function:	<i>show.param</i> displays all positional parameters
Note:	If the parameters contain metacharacters for file name expansion, the command displays all matching file names.

Table 41: Summary of the *show.param* shell script

Keyword parameters

Another type of variable is the keyword parameter. You use the following standard syntax to assign a value to a keyword parameter:

```
keyword_parameter=value
```

Make sure that you do not enter a blank on either side of the equals sign (=). In the following example, *var1* is a keyword parameter and *myname* is the value you assign to it:

```
var1=myname
```

You use the dollar symbol (\$) in front of a variable name to reference the value of this variable. In the above example, the variable *var1* tells the shell to substitute the value *myname* (assigned to the variable *var1*) for any occurrence of the string *\$var1*.

The first character of a variable name must be a letter or an underscore. The rest of the name can consist of letters, underscores and digits. Like shell script files, variables should not have the same names as system commands. Furthermore, the shell reserves some variable names, listed below, that you may not use for your variables:

CDPATH	defines the search path for the <i>cd</i> command.
HOME	is the default variable for the <i>cd</i> command (<i>home</i> directory).
IFS	defines the internal field separators (normally the blank, tab and carriage return).
LOGNAME	is your login name.
MAIL	specifies the file that contains your electronic mail.
PATH	defines the search path used by the shell to find commands.
PS1	defines the primary prompt (default: \$).
PS2	defines the secondary prompt (default: >). The character defined by <i>PS2</i> appears at the start of a continuation line if your command input is longer than one line.
TERM	specifies the terminal type. This value must be set before you can work with <i>vi</i> .
TERMINFO	specifies the directory to be searched for information about your terminal (e.g. its screen size).
TZ	defines the time zone (default: <i>EST5EDT</i>)

Table 42: Keyword parameters (default variables)

Many of these variables are described in section 5.4 *Modifying your login environment*. For further information, see *sh* in the *Commands* manual [5].

There are two ways of seeing what values are assigned to your shell variables. Firstly, you can use the following command line to display the value of a single variable:

```
echo $variable_name
```

The system displays the value of *variable_name*. Secondly, you can list the values of all shell variables using the *env* command (see *env* in the *Commands* manual [5]). You simply enter *env* on a command line by itself. The system displays a list of the variable names and their values.

You can obtain even more detailed information on the values of your shell variables using the *set* command. *set* also includes the values of variables that have not been exported in its list (see *set* in the *Commands* manual [5]).

5.3.3 Assigning a value to a variable

During a *vi* session (or a session with another editor), you can set the *TERM* variable (in this case to the 97801 Terminal) using the following command line:

```
TERM=97801
export TERM
```

This is the simplest way to assign a value to a variable. However, there are several other ways of doing this. For example, you can

- assign the input of the *read* command to a variable
- use the output of a command as the value of a variable (command substitution)
- assign values using positional parameters

The following sections describe these options in detail.

The read command

You use the *read* command within a shell script to prompt the user for values for variables. This command has the following standard syntax:

```
read variable(s)
```

You can tell the script to display instructions (in the following example *Please enter ...*) by specifying the *echo* command before the *read* command in a script. The *read* command will wait until the user has entered a string and pressed , and will then make that string the value of the variable.

The following example illustrates how to write a simple shell script called *num.please* that enables you to retrieve numbers from a telephone list and display them on the screen. The script contains the following commands:

echo	to prompt the user for a person's last name
read	to assign the input value to the variable <i>name</i>
grep	to search the <i>list</i> file for this variable

The finished script should look like this:

```
$ cat num.please
echo Please enter last name:
read name
grep $name $HOME/list
$
```

The next example is a script called *mknum* that enables you to create a telephone list. If you want to add the output of the *echo* command to the end of the telephone list *list*, you must use `>>` to redirect it. If you used `>` to redirect output, *list* would contain only the last telephone number you entered.

Use the *cat* command to view the contents of the *mknum* script:

```
$ cat mknum
echo Please enter name
read name
echo Please enter telephone number
read num
echo $name $num >> list
$
```

The final *echo* command entered the values of the variables *name* and *num* in the *list* file.

Now grant execute permission for the *mknum* script:

```
$ chmod u+x mknum
$
```

In the next example, *mknum* creates a new entry for *Mr. Cooper*. *num.please* then displays the telephone number of *Mr. Cooper*:

```
$ mknum
Please enter name
Mr. Cooper
Please enter telephone number
631-43572
$ num.please
Please enter name
Cooper
Mr. Cooper 631-43572
$
```

Note that you can specify either *Mr. Cooper* or simply *Cooper* as the value of the *name* variable.

Tables 43 and 44 summarize the shell scripts *mknum* and *num.please*

Summary	
Shell script <i>mknum</i> - place name and telephone number on a telephone list	
Command	Arguments
mknum	Entered interactively
Function:	This script prompts you to enter the name and telephone number of a person and adds that name and number to your telephone list.
Note:	This is an interactive command.

Table 43: Summary of the mknum shell script

Summary	
Shell script <i>num.please</i> - display a person's name and telephone number	
Command	Arguments
num.please	Entered interactively
Function:	This script prompts you to enter a person's last name and then displays the person's full name and telephone number.
Note:	This is an interactive command.

Table 44: Summary of the num.please shell script

Substituting command output for the value of a variable

You can substitute a command's output for the value of a variable by using *command substitution*. Command substitution has the following syntax:

```
variable='command'
```

The output of *command* is used as the value of *variable*.

In section 5.2 *The shell command language*, the *date* command was piped to the *cut* command to obtain the current time. The command line looked like this:

```
$ date | cut -c12-19
```

You can also incorporate this line in a simple shell script called *t* that tells you the time.

```
$ cat t
time=`date | cut -c12-19`
echo The time is: $time
$
```

Make sure that you do not enter a blank on either side of the equals sign (=).
Make the file executable:

```
$ chmod u+x t
```

You now have a script that tells you the time. Call your script:

```
$ t
The time is: 10:36:53
$
```

Table 45 summarizes the shell script *t*.

Summary	
Shell script <i>t</i> - display the current time	
Command	Argument
<i>t</i>	none
Function:	<i>t</i> displays the time in hours, minutes and seconds.

Table 45: Summary of the *t* shell script

Assigning values with positional parameters

You can assign a positional parameter to a keyword parameter using a command line with the following syntax:

```
var1=$1
```

The next example is a simple script called *simp.p* that assigns a positional parameter to a variable. View the contents of the *simp.p* file:

```
$ cat simp.p
var1=$1
echo $var1
$
```

Of course, you can also assign the output of a command that uses positional parameters as the value of a variable:

```
$ person='who | grep $1'
```

The script called *log.time* in the next example logs the results of the *whoson* script. The output of the *whoson* script is assigned to the *person* variable and entered in a file called *login.file* using the *echo* command. The final *echo* command displays the value of *\$person*, which is the same as the output from the *whoson* script:

```
$ cat log.time
person=`who | grep $1`
echo $person >> $home/login.file
echo $person
$
```

Now call the *log.time* script:

```
$ log.time maryann
maryann      tty61          Apr 11 10:26
$
```

Table 46 summarizes the shell script *log.time*

Summary	
Shell script <i>log.time</i> - log and display a login name (user-defined)	
Command	Arguments
<i>log.time</i>	<i>login_name</i>
Function:	If the specified user is currently logged in, the shell script <i>log.time</i> enters the output from the <i>who</i> command in a file called <i>login.file</i> and displays it on your terminal.

Table 46: Summary of the *log.time* shell script

5.3.4 Shell programming constructs

The shell programming language has several constructs that give added flexibility to your programs:

- *Comments* (identified by #) let you document the individual features of a script.
- The *here document* allows you to include in the shell script itself lines intended as input for a command in the shell script.
- The *exit* command lets you terminate a script at a point other than the end of the script and define return codes.

- *for* and *while* loops instruct a script to repeat a set of commands several times.
- The conditional statements *if* and *case* instruct the script not to execute a group of commands unless certain conditions are met.
- You can use the *break* command to terminate a loop unconditionally.

The following sections discuss these constructs in detail.

Comments

Comments are introduced by a hash symbol (#). The shell ignores all text on a line between the hash and the next newline, entered using `\n`. If the hash appears at the start of the line, the entire line is interpreted as a comment. If the hash appears after a command, the command is executed and the rest of the line is ignored. The newline always ends a comment. A comment line has the following basic format:

```
#comment
```

The following lines in a shell script will be ignored when the script is executed:

```
# This script works out the date up to  
# the year 2001. Leap years are identified  
# by an asterisk (*).
```

In the next example, only the *pwd* command is executed. The shell interprets the *ls* command as a comment:

```
$ pwd; #ls  
/usr2/guide
```

The functions of programs or scripts are always easier to follow if they contain comments. By using comments, you will make your programs much easier to read and maintain.

Here documents

Shell scripts use here documents for commands that read from standard input. The standard input of the command is redirected to the here document. A here document consists of the redirection symbol `<<` and a delimiter that specifies the start and end of the input lines. The delimiter can be a character or a characters string and is often the exclamation mark (!).

A here document has the following standard syntax:

```
command << delimiter
...input lines...
delimiter
```

In the next example, the *gbday* script uses a here document to send a generic birthday greeting by redirecting lines of input to the *mail* command:

```
$ cat gbday
mail $1 <<!
Best wishes to you on your birthday, Laura Palmer.
Regards, Bob
!
$
```

You enter the following command to send the greeting to the user *laura*. You must specify the login name as an argument:

```
$ sh gbday laura
$
```

The next time *laura* reads her mail, she will receive the following greeting:

```
$ mail
From bob Wed May 14 14:31 1992
Content-Length: 51

Best wishes to you on your birthday, Laura Palmer.
Regards, Bob.

$
```

Table 47 summarizes the syntax and function of the shell script *gbday*.

Summary	
Shell script <i>gbday</i> - send a generic birthday greeting (user-defined)	
Command	Arguments
post	<i>login_name</i>
Function:	<i>gbday</i> sends a generic birthday greeting to the owner of the login name you specify as an argument.

Table 47: Summary of the *gbday* shell script

Return codes

Most shell commands output return codes that tell you whether the command executed normally. The return code *0* (zero) indicates that the command did execute normally. Any other value indicates that errors occurred during processing. The return code is not displayed automatically; you must request it explicitly using the special parameter *\$?*.

Checking the return code

After executing a command interactively, you can check its return code by entering

```
$ echo $?
```

Consider the following example:

```
$ cat hi
This is a file called hi
$ echo $?
0
$ cat hello
cat: cannot open hello
$ echo $?
2
$
```

In the first case, your directory contains the file called *hi* for which you have read permission. The *cat* command executes normally, displaying the contents of the file on the screen and outputting the return code *0*, which you can check using the *\$?* parameter. In the second case, either the file does not exist or you do not have read permission for it. *cat* issues a message indicating the cause of the error and outputs the return code *2*.

Using return codes with the *exit* command

A shell script does not normally terminate until the last command in the script has executed. However, you can use the *exit* command to abort a script at some other point. Perhaps more importantly, you can also use the *exit* command to set a return code of between 0 and 255. For further information on the *exit* command, see the *Commands* manual [5].

The shell's garbage can: the */dev/null* file

The file system contains a file called */dev/null*. You can tell the shell to write any unwanted output to this file, thus discarding it. Try entering the following command line:

```
who > /dev/null
```

The result of the *who* command, i.e. the names of all users who are logged in, is thrown immediately into the shell's garbage can.

Loops

In the previous examples in this chapter, the commands in the shell scripts have been executed in sequence. You can use *for* and *while* loops to instruct a shell script to execute a set of commands several times.

The for loop

The *for* loop executes a sequence of commands once for each entry in a list. The *for* loop has the following standard syntax:

```
for variable
  in list_of_values
do
  command1
  command2
  .
  .
  last command
done
```

For each iteration of the loop, the next item in the list is assigned to the variable specified in the *for* clause. References to that variable may be made anywhere in the commands within the *do* clause.

It is easier to read a shell script if the loops have a visually clear structure. Since the shell ignores blanks at the beginning of lines, you can indent each section of commands as shown above. This also makes it easier for you to check whether each *do* has a corresponding *done* at the end of the loop.

You can assign the variable any name. If, for example, you call it *var*, then the values given in the list after the keyword *in* will be assigned in turn to *var*. References in the command list to *\$var* will make the value available. If there is no *in* clause, the values for *var* will be all arguments specified in the command line and assigned to the special parameter *\$**. The command list between the keywords *do* and *done* will be executed once for each value.

Once the commands have been executed for the last value in the list, the script executes the next line below *done*. If there are no more lines here, the script is terminated.

Create a script that moves files to another directory. This script should include commands for the following functions:

echo	to prompt the user for the path name of the new directory
read	to assign the path name to the <i>PATH</i> variable
for variable	to give the variable the name <i>file</i> ; it can be referenced using <i>\$file</i> in the command sequence
in <i>list_of_values</i>	to specify a list of values If there is no <i>in</i> clause, the list of values is assumed to be the value of <i>\$*</i> (all arguments entered on the command line)
do <i>commands</i>	to specify a command sequence. This script uses the following statement for this purpose: <i>do</i> <i>mv \$file \$path/\$file</i> <i>done</i>

View the contents of the shell script *mv.file*. This script already contains the values for the *file* variable:

```
$ cat mv.file
echo Please enter the directory path
read path
for file
  in memo1 memo2 memo3
do
  mv $file $path/$file
done
$
```

If you want to use different files each time you call the script, you can assign the values using positional parameters or the *read* command. If you use positional parameters, you can omit the keyword *in*, as shown in the modified shell script *mv.file* below:

```
$ cat mv.file
echo Please enter the directory path
read path
for file
do
  mv $file $path/$file
done
$
```

You can move several files to a different directory at the same time by specifying a list of file names as the argument of the command.

Table 48 summarizes the shell script *mv.file*.

Summary	
Shell script <i>mv.file</i> - move files to another directory (user-defined)	
Command	Arguments
<i>mv.file</i>	<i>file_names</i> interactive
Function:	Moves files to a new directory.
Note:	You must specify file names as arguments for this script. The script prompts you to enter the path of the new directory.

Table 48: Summary of the *mv.file* shell script

The while loop

Another loop statement, the *while* loop, uses two groups of commands. The shell executes the command sequence in the first group (in the *while* clause) to check whether a specified condition is true. The shell continues to execute the command sequence in the second group (the *do...done* clause) as long as the condition is true.

The *while* loop has the following standard syntax:

```
while
  command1
  command2
  .
  .
  last command
do
  command1
  command2
  .
  .
  last command
done
```

Let us assume that a shell script called *enter.name* uses a *while* loop to write a list of names to a file. The script contains the following command lines:

```
$ cat enter.name
while
  read x
do
  echo $x>>xfile
done
$
```

With some added refinements, the script looks like this:

```
$ cat enter.name
echo Please enter one or more names and press return.
echo Please end the list of names by pressing Control d
while read x
do
  echo $x>>xfile
done
echo xfile contains the following names:
cat xfile
$
```

Once you have pressed **CTRL** d to indicate that you have finished entering names, the script executes the commands below *done*.

Since you used metacharacters in the first two *echo* command lines, you must escape these characters using single quotes ('...') or double quotes ("..."). The *enter.name* script produces the following results:

```
$ enter.name
Please enter one or more names and press return.
Please end the list of names by pressing Control d
Dale B. Cooper
Laura
Audrey

CTRL d
xfile contains the following names:
Dale B. Cooper
Laura
Audrey
$
```

Once the loop has terminated, the script displays all the names contained in the *xfile* file.

The until loop

The *until* statement checks whether the condition specified in *until* is *false* (not zero). As soon as the condition becomes *true*, the *until* statement terminates.

The *until* loop has the following standard syntax:

```
until
command1
command2
    .
    .
    last command
do
command1
command2
    .
    .
    last command
done
```

Let us assume that you want to delete all the empty files from a directory. Provided you are the owner of the directory and the files involved, you can use a shell script to perform this operation. Create a shell script called *delete* with the following contents:

- a *for* statement. This makes each entry in the current directory available to the *until* statement for checking.
- the *test* command with the option *-s*. This checks whether a file exists and whether or not it is empty. If the file is empty, this command issues the exit status 1 (*false*).
- the *rm* command. This deletes the file read by *\$x*.
- the *echo* and *break* commands. *echo* issues a message telling you which file has been deleted and *break* terminates the *until* loop.
- The *for* command then reads the next entry and passes it to the *until* statement for checking.

Now create the script:

```
for x in *
do until test -s "$x"
do
    rm $x
    echo File $x deleted
    break
done
done
```


Conditional statements

if...then

The *if* statement instructs the shell script not to execute the *then* sequence of commands unless the final command in the *if* command list returns the status *true*. You terminate an *if* statement with the keyword *fi*.

The *if...then* statement has the following standard syntax:

```
if
command1
command2
.
.
last command
then
command1
command2
.
.
last command
fi
```

The shell script *search* below illustrates how to use the *if...then* statement. The *search* script uses the *grep* command to search for a word in a file. Once *grep* has found the word, the script issues an appropriate message. Enter the *search* script at your terminal and test it yourself.

```
$ cat search
echo Enter the word and the file name.
read word file
if grep $word $file
then echo $word is in $file
fi
$
```

The *read* command assigns values to the variables *word* and *file*. The first characters you enter before a blank are assigned to the *word* variable. The rest of the characters (including the blank) are assigned to the *file* variable.

if...then...else

Within an *if...then* statement, you can also use *else* to specify an alternate command list. The script executes these commands if the *if* command list returns the status *false*. The *if...then...else* statement has the following standard syntax:

```
if
command1
command2
.
.
last command
then
command1
command2
.
.
last command
else
command1
command2
.
last command
fi
```

You can now improve your *search* script so that it will always issue a message, regardless of whether it can find the word. The improved script looks like this:

```
$ cat search
echo Enter the word and the file name.
read word file
if
  grep $word $file >/dev/null
then
  echo $word is in $file
else
  echo $word is NOT in $file
fi
$
```

Table 49 summarizes the improved *search* script.

Summary	
Shell script <i>search</i> - searches for a word in a file and displays the results (user-defined)	
Command	Arguments
search	interactive
Function:	Searches for a word in a file and tells you whether the file contains this word.
Note:	The command prompts you to enter the arguments, i.e. the word and the file name.

Table 49: Summary of the search shell script

Testing conditions using test

The *test* command, which checks whether certain conditions are true or not, is useful in conditional statements. If a given condition is true, the loop continues. If the condition is false, the loop terminates and the next command is executed. Some of the options for the *test* command are:

test -r <i>file</i>	True if the file exists and you have read permission.
test -w <i>file</i>	True if the file exists and you have write permission.
test -x <i>file</i>	True if the file exists and you have execute permission.
test -s <i>file</i>	True if the file exists and contains at least one character.
test <i>var1</i> -eq <i>var2</i>	True if <i>var1</i> is equal to <i>var2</i> .
test <i>var1</i> -ne <i>var2</i>	True if <i>var1</i> is not equal to <i>var2</i> .

Let us assume that you want to use a shell script to move all executable files from the current directory to your *bin* directory. You can use the *test -x* command to select the executable files. Review the following example of a *for* statement that occurs in the *mv.file* script:

```
$ cat mv.file
echo Please enter the directory path
read path
for file
do
    mv $file $path/$file
done
$
```

Create a script called *mv.ex* that includes an *if test -x* statement in the *do...done* loop for selecting executable files. Your script will look like this:

```
$ cat mv.ex
echo Please enter the directory path
read path
for file
do
    if test -x $file
    then
        mv $file $path/$file
    fi
done
$
```

The directory path is the path from the current directory to the *bin* directory. Now use the shell variable *\$HOME*. This saves having to enter the entire path again, since *\$HOME* defines the access path up to your home directory. *\$HOME/bin* defines the path to your *bin* directory, which is a subdirectory of your home directory.

Modify *mv.ex* so that it does not prompt you to enter the directory name. You can omit the *path* variable:

```
$ cat mv.ex
for file
do
    if test -x $file
    then
        mv $file $HOME/bin/$file
    fi
done
$
```

Test the command on all files in the current directory, specifying the asterisk (*) as the command argument. The command line in the following example executes the command in the current directory, changes to the *bin* directory and then lists the files in this directory. This directory should now contain all the executable files.

```
$ mv.ex *
$ cd; cd bin; ls
list_of_all_executable_files
$
```

Table 50 summarizes the syntax and function of the shell script *mv.ex*.

Summary	
Shell script <i>mv.ex</i> - move all executable files from the current directory to the <i>bin</i> directory	
Command	Arguments
mv.ex	List of files in the current directory
Function:	Moves all executable files from the current directory to the <i>bin</i> directory.
Note:	The executable files in the <i>bin</i> directory (or any directory specified in the <i>path</i> variable) can be executed from any directory.

Table 50: Summary of the mv.ex shell script

case...esac

The *case...esac* statement allows you to choose one of several patterns. You can specify what commands you want to be executed for each pattern specified. The keywords *case*, *in* and *esac* are mandatory.

The pattern section must begin with the keyword *in*; you must place a right parenthesis *)* after the last character of each pattern. Each command or command sequence within a pattern must end in two semicolons *;;*. You end the *case* construct with the keyword *esac* (the letters of the word *case* reversed).

The *case* statement has the following standard syntax:

```
case word
in
  pattern1)
    command line1
    .
    last command line
  ;;
  pattern2)
    command line1
    .
    last command line
  ;;
  *)
    command line1
    .
    last command line
  ;;
esac
```

The *case* statement first compares the *word* specified after *case* with *pattern1* in the first pattern section. If they match, the script executes the command lines between the first pattern and the corresponding semicolons (;:).

If *word* does not match the first pattern, the script proceeds to the second pattern. Once the script finds a pattern that matches *word*, it does not try to match any more patterns, but goes to the command following *esac*.

The *** character used as a pattern matches any *word*. You can specify commands for this pattern that you want the script to execute if none of the other patterns match *word*. You must specify this pattern as the last pattern in the *case* statement so that the script checks the other patterns first. This helps you detect incorrect or unexpected input.

The patterns you specify in the *pattern* part of each section can also contain the metacharacters ***, *?* and *[]*, increasing the flexibility of the statement.

The *set.term* script contains a good example of a *case...esac* statement. This script sets the shell variable *TERM* to the type of terminal you are using. You use the following command line to execute this script:

```
$ TERM=terminal_name
```

In the following example, the terminal is a Teletype 4420, 5410 or 5420.

The *set.term* script first checks whether the value *4420* has been entered for *term*. If so, it assigns the value *T4* to the *TERM* variable. Otherwise the script checks for the other two values (*5410* and *5420*). Once the script has executed the commands under the first matching pattern, it goes to the first command after *esac*.

The last pattern is ***, which stands for any input. The system issues a message to inform you that there is no pattern for the specified terminal and allows you to terminate the *case* statement:

```
$ cat set.term
echo If you have a terminal of type TTY 4420, enter: 4420
echo If you have a terminal of type TTY 5410, enter: 5410
echo If you have a terminal of type TTY 5420, enter: 5420
read term
case $term
in
    4420)
        TERM=T4
        ;;
    5410)
        TERM=T5
        ;;
    5420)
        TERM=T7
        ;;
    *)
        echo invalid terminal type
        ;;
esac
export TERM
echo end of program
$
```

Note the use of the *export* command in the above example. You use this command to make a variable within your environment available to other shell scripts. What would happen if you specified *** as the first pattern and *echo invalid terminal type* as the command line? The *set.term* script would never assign a value to *TERM*, since the first pattern, i.e. any terminal, would always match.

Table 51 summarizes the syntax and function of the shell script *set.term*:

Summary	
Shell script <i>set.term</i> - assign a value to <i>TERM</i> (user-defined)	
Command	Arguments
set.term	interactive
Function:	Assigns a value to the shell variable <i>TERM</i> and then exports that value, making it available to other shell scripts.
Note:	This command prompts you to enter a particular terminal type to be used as a search pattern in the <i>case</i> statement.

Table 51: Summary of the set.term shell script

Unconditional control statements: the break and continue commands

The *break* statement terminates the current loop unconditionally. The script then proceeds to the commands that follow the *done*, *fi* or *esac* statement. If there are no commands after *done*, *fi* or *esac*, the script terminates.

In the sample program *set.term*, you could have used the *break* command in place of the *echo* command to terminate the script:

```
$ cat set.term
echo If you have a terminal of type TTY 4420, enter: 4420
echo If you have a terminal of type TTY 5410, enter: 5410
echo If you have a terminal of type TTY 5420, enter: 5420
read term
case $term
    in
        4420)
            TERM=T4
            ;;
        5410)
            TERM=T5
            ;;
        5420)
            TERM=T7
            ;;
        *)
            break
    ;;
esac
export TERM
echo end of program
$
```


The continue command

The *continue* command causes the script to go to the next iteration of the *while* or *for* loop immediately, without executing the rest of the commands in the loop.

5.3.5 Debugging shell scripts

The *sh* command has options that help you find and correct errors in a shell script. You will become familiar with two of these options in this section. For further information, see *sh* in the *Commands* manual [5].

<code>sh -n shell_script_name</code>	The subshell reads and interprets all commands but does not execute them.
<code>sh -v shell_script_name</code>	The subshell first writes all the commands you enter to the screen. Only then does it execute the corresponding command.

Let us assume that you want to use the *-n* and *-v* options to check your new shell script *directory* for syntax errors without executing the commands contained in it. Note: you have forgotten to include parentheses in the definition of the *ll* function.

```
$ cat directory
set `echo *`
echo `pwd`:
ll {
  ls -al $* | pg
}
ll
$
```

The third line of the function is incorrect and should read:

```
ll() {
```

Now use the *-n* and *-v* options to check the syntax of the shell script:

```
$ sh -nv directory
set `echo *`
echo `pwd`:
ll {
  ls -al $* | pg
}
ll
directory: syntax error at line 6: '}' unexpected
$
```

Because you set `-v`, the shell first outputs each command line it reads, followed by the error message in the last line.

`-n` prevents the shell from executing the individual commands. If you call the *directory* script with `-n` only, the shell outputs only the error message.

You can use the *tee* command to check a pipeline for errors. This command has the following standard syntax:

```
command1 | tee backup_file | command2
```

If, for example, you want to know how many users are logged in, you simply enter:

```
$ who | wc -l
4
$
```

If you now want to check whether the output 4 is correct, you simply use the *tee* command in conjunction with an output file. If it does not already exist when you call the *tee* command, the output file is created automatically.

```
$ who | tee security | wc -l
4
$
```

The same results as before appear on the screen. Using *tee*, however, had the following effect: *tee* read data from standard input, i.e. the output of the *who* command, and wrote this data to a file called *security*. The following command *wc -l* then used the contents of the *security* file as its input. You simply use the *cat* command to see what data has been written to the *security* file.

```
$ cat security
andrea  tty01  Dec 05 08:08
william tty03  Dec 05 09:15
felix   tty07  Dec 05 09:17
cooper  tty06  Dec 05 08:16
$
```

You can see that four users were logged in. For further information on the *tee* command, see *tee* in the *Commands* manual [5].

5.4 Modifying your login environment

Reliant UNIX provides you with various options for customizing your login environment. For example, you can add commands to the *.profile* file, set terminal options and use reserved shell variables. This section describes some of the options available to you.

5.4.1 The *.profile* file

What happens when you log in?

When you log in, the shell first examines the *.profile* file, provided that it is located in your home directory. This file is also called the login script and contains commands that control your shell environment.

The *.profile* file is a shell script that you can edit yourself. On some systems, however, it must be edited by the system administrator. You can enter the following command line to check whether your home directory contains a file called *.profile*:

```
$ ls -al $HOME
```

Before you edit *.profile*, make a backup copy of it, under the name *save.profile*, for example. Do not make any changes to *.profile* until you have done so. This ensures that you always have the original to fall back upon if necessary. Now create the backup copy:

```
$ cp .profile save.profile
```

You can add commands to your *.profile* login script just as you would to any other shell script. You can also use the *stty* command to configure different terminal options and set certain shell variables. See *Setting terminal options* further on in this section.

Adding commands to the *.profile* file

Call *.profile* using an editor and append the following *echo* command:

```
echo Good morning! I've been waiting for you.
```

Save the text and quit the editor. If you want the changes you have made to your *.profile* login script to be valid during the current Reliant UNIX session, you must call *.profile* with the dot (.) shell command. This causes the shell to read the commands in the login script and initialize your environment accordingly. Try entering the following command:

```
$ . .profile
```

The system should display the following message:

```
Good morning! I've been waiting for you.
```

Setting terminal options

You can use the *stty* command to set up your shell environment according to your own requirements. You can specify the following options with the *stty* command:

-tabs	This option outputs tabs unchanged. It expands the tab setting to eight blanks by default; however, you can change the number of blanks for each tab. For further information, see <i>stty</i> in the <i>Commands</i> manual [5].
echoe	This option causes characters to be deleted from the screen when you press the #T19 key.

If you want to use these options of the *stty* command, you can insert the corresponding command lines in your *.profile* login script just as you would insert lines in any shell script. The last three lines of your *.profile* file might then look like this:

```
echo Good morning! I've been waiting for you.  
stty -tabs  
stty echoe
```

Shell variables

You can redefine many reserved shell variables in your *.profile* login script. You can use the following command to display the current value of any shell variable, in this case *\$TERM*:

```
$ echo $TERM
97801
$
```

This section describes four of the most important shell variables: *HOME*, *PATH*, *TERM* and *PS1*:

HOME The path name of your home directory is assigned to this variable. Use the *cd* command to move to your home directory. If you do not specify a directory, *cd* moves you to the current *\$HOME* directory. Enter the following command:

```
$ echo $HOME
```

Note the output and then enter the following command:

```
$ pwd
```

Although you obtained the same result in each case, these two commands each had a different effect. *echo \$HOME* displayed your home directory, while *pwd* displayed the directory in which you were currently working. If, when you enter *pwd*, you are working in a subdirectory, the command displays the name of the current subdirectory.

PATH The search path for finding and executing commands is assigned to this variable. You can see the current values for *PATH* by entering the *echo \$PATH* command. The system responds by displaying the current value of your *PATH* variable.

```
$ echo $PATH
:/mylogin/bin:/bin:/usr/bin
$
```

A colon on its own (in this case, the first colon) stands for the current directory. If you enter a command, therefore, the shell searches for the command in the current directory first. If the shell does not find it there, it searches in the rest of the directories specified in the *PATH* variable. In this case, it searches in turn through the directories */mylogin/bin*, */bin* and */usr/bin*. A colon that follows a directory name acts as a delimiter, separating the different path names in the string specified for *\$PATH*. Note that the colon that represents the current directory does not necessarily have to be at the start of the path specification. You could, for example, specify the following path:

```
$ echo $PATH
/mylogin/bin:/bin::/usr/bin
$
```

The shell would then look for a command in the *mylogin/bin* directory first, then in */bin*, then in the *current directory* and finally in */usr/bin*.

If you are working on a project with several other people, you can create a *bin* directory for the group. This directory can contain special shell scripts that are used only by the people working on the project. The path might be named *\$HOME/project1/bin*. To do this, edit your *.profile* file and add *\$HOME/project1/bin* to the end of the path specified for the *\$PATH* variable:

```
$ PATH=$HOME/project1/bin
```

TERM You use this variable to tell the shell what type of terminal you are using. To assign a value to it, you must execute three commands in sequence. For example, you assign a 97801 terminal as follows:

```
TERM=97801
export TERM
tput init
```

The first two lines tell the computer what type of terminal you are using. The third line (containing the *tput* command) initializes the terminal for the terminal type you have set. This command must always be specified after the variable has been exported.

If you do not want to set the *TERM* variable every time you log in, add the above command lines to your *.profile* login script.

If you work with several different types of terminal, your *.profile* login script should also contain a script like *set.term* (see Table 51: *Summary of the set.term shell script* and the example for *set.term* that follows it).

PS1 The primary prompt (default: *\$*) is assigned to this variable. You can define a new prompt by changing the *PS1* variable in your *.profile*.

If, as in the following example, your prompt consists of more than one word, you must enclose it in double quotes (*"*). To do this, assign the following value to this variable in your *.profile*:

```
$ PS1="Your wish is my command"
```

Now execute your *.profile* login script using the dot command and watch for your new prompt.

```
$ . .profile
Your wish is my command
```

The default prompt, the dollar sign \$, does not reappear until you change the *PS1* variable in your *.profile* accordingly.

5.5 Exercises

5.5.1 Command language exercises

1-1 What happens if you use an asterisk (*) at the start of a file name? Try to list some of the files in your directory by specifying the asterisk * together with the last letter of a file name. What happens?

1-2 Enter the following commands. What output do you obtain for each command?

```
$ cat [0-9]*
```

```
$ echo *
```

1-3 Can you specify the question mark (?) at the start or in the middle of a character string? Try it.

1-4 Do the names of any of your files start with a number? How can you list these without listing the rest of the files in your directory? List only those files that start with a lowercase letter between a and m. (Hint: use the square brackets (*[]*) to specify a range of numbers or letters.)

1-5 Can you place a command in background mode on a line that is executing several other commands sequentially? What happens? (Hint: use ; and &). Can the command that is executing in the background appear at any position on the command line? Try entering the command at different positions on the command line. Experiment with each new character to see what it can do.

1-6 Redirect the output of *pwd* and *ls* into a single file.

Remember to use >> (append) for the second redirection, since you want to redirect both commands to the same file. Otherwise, the output from the *pwd* command will be overwritten.

1-7 You have already seen how to extract the time from the output of the *date* command. Now try redirecting the date only (without the time) to the *banner* command. What do you need to change in the following command line?

```
$ banner 'date | cut -c12-19'
```


5.5.2 Shell programming exercises

- 2-1 Create a shell script called *time* from the following command line:
- ```
$ banner 'date | cut -c12-19'
```
- 2-2 Write a shell script that will display a banner containing only the date. Be careful not to give your program the same name as a Reliant UNIX command.
- 2-3 Write a shell script that will send a message to different users on your system.
- 2-4 Redirect the output of the *date* command to a file, excluding the time and year specifications.
- 2-5 Use the *echo* command to write the words *Dear friends* to the above file without deleting the date.
- 2-6 Using the above exercises, write a shell script called *send.memo* that will send a memo to the same users as in Exercise 2-3. Your file should contain:
- the current time and the words *Dear friends* at the top of the memo
  - the text of the memo (that you have already created in a file called *memo*)
  - the closing sentence
- 2-7 How can you read variables into the *mv.file* program using the *read* command?
- 2-8 Use a *for* loop to move several files from your current directory to another directory. How can you move all your files to another directory?
- 2-9 How can you modify the *search* program so that it will search through more than one file?
- Hint:** for file  
in \$\*
- 2-10 Change your primary prompt to the word *Hello*.
- 2-11 Check the settings of the variables *\$HOME*, *\$TERM* and *\$PATH* in your environment.

## 5.6 Solutions

### 5.6.1 Solutions to the command language exercises

- 1-1 You use the asterisk (\*) at the start of a character string to refer to all file names that end in this string. This includes the file name that consists only of the string.

```
$ ls *t
123t
new.t
t
$
```

- 1-2 The *cat* [0-9]\* command concatenates and displays the contents of all files that have names starting with a number.

The *echo* \* command lists all the files in the current directory.

- 1-3 You can specify the question mark (?) at any position in a file name.

- 1-4 The *ls* [0-9]\* command lists only those file names that start with a number.

The *ls* [a-m]\* command lists only those file names that start with a lowercase letter in the range *a* through *m*.

- 1-5 When a command on a sequential command line is executed in the background, the system immediately displays the process ID of the job.

The ampersand (&) must always appear at the end of the command line.

- 1-6 Your command line should look like this:

```
$ cd ; pwd > junk ; ls >> junk
```

- 1-7 Change the -c option so that the command line looks like this:

```
banner 'date | cut -c1-10'
```

## 5.6.2 Solutions to the shell programming exercises

2-1 Use the *cat* command to view the contents of your shell script:

```
$ cat time
banner `date | cut -c12-19`
$
```

Make *time* an executable file, then invoke it:

```
$ chmod u+x time
$ time
$
```

The script will display a banner containing the current time.

2-2 The following script displays a banner containing the date:

```
$ cat mydate
banner `date | cut -c1-10`
$
```

2-3 You can use the following shell script to send a message to any user on your system:

```
$ cat tofriends
echo Type in the name of the message file.
read note
mail laura audrey bob dale < $note
$
```

If you used parameters to represent the login names, your script may look like this:

```
$ cat tofriends
echo Type in the name of the message file.
read note
mail $* < $note
$
```

2-4 Enter the following command line:

```
$ date | cut c1-10> file1
```

2-5 Enter the following command line:

```
$ echo Dear friends >> file1
```

**2-6 Your shell script should look like this:**

```
$ cat send.memo
date | cut c1-10 > memo1
echo Dear friends >> memo1
cat memo >> memo1
echo A message from Windom Earle >> memo1
mail laura audrey bob dale < memo1
$
```

**2-7 View the contents of *mv.file*:**

```
$ cat mv.file
echo Please type in the path name
read path
echo Please type in the file name(s) and end with
Control d
while
read file
do
 mv $file $path/$file
done
echo all done
$
```

**2-8 View the contents of *mv.file*:**

```
$ cat mv.file
echo Please type in the path name
read path
for file in $*
do
 mv $file $path/$file
done
$
```

The following command line moves all the files in the current directory:

```
$ mv.file *
```

**2-9 See the hint given in Exercise 2-9.**

```
$ cat search
for file
in $*
do
 if grep $word $file >/dev/null
 then echo $word is in $file
 else echo $word is NOT in $file
 fi
done
$
```

- 2-10 Change the primary prompt to *Hello* by adding the following commands to your *.profile* login script:

```
PS1=Hello
export PS1
```

- 2-11 To check the variable settings for your login environment, enter

```
$ echo $HOME

$ echo $TERM

$ echo $PATH
```

# 5.7 Summary of the shell command language

This section summarizes the shell command language and shell programming constructs described in this chapter. For a complete overview, see the *Commands* manual [5].

## Shell metacharacters

| Character   | Function                                                                                                                                                      |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| * ? [ ] [-] | The asterisk, question mark and square brackets provide a shortcut for specifying file names using search patterns.                                           |
| &           | The ampersand places commands in background mode, leaving your terminal free for other tasks.                                                                 |
| ;           | The semicolon separates commands specified on the same command line.                                                                                          |
| \           | The backslash escapes metacharacters such as *, ?, [ ], &, ;, >, < and  .                                                                                     |
| ' ...'      | Single quotes cancel the meaning of the blank as a delimiter and escape all metacharacters.                                                                   |
| "..."       | Double quotes cancel the meaning of the blank as a delimiter and escape all metacharacters except \$ and '.                                                   |
| >           | The greater than sign redirects the output of a command to a file (overwriting the original contents of that file).                                           |
| <           | The less than sign redirects input so that a command reads its input from a file.                                                                             |
| > >         | The double greater than sign appends the output of a command to an existing file.                                                                             |
|             | The pipe symbol causes the output of one command (to the left of the pipe symbol) to be used as input for the next command (to the right of the pipe symbol). |
| ` ...`      | Used in pairs, backquotes substitute the output of a command for the argument in a command line.                                                              |
| \$          | The dollar sign is used in positional parameters and user-defined variables; it is also the default shell prompt.                                             |

Executing and terminating processes

| Command                 | Function                                                                                                                                                                        |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| at                      | executes the commands that follows it at the specified ctime. You end the command with <code>CTRL d</code> .                                                                    |
| at -l                   | outputs information on the jobs that are currently in the <i>at</i> or <i>batch</i> queue.                                                                                      |
| at -r                   | removes the job issued with <i>at</i> or <i>batch</i> from the queue.                                                                                                           |
| batch                   | places the commands that follow it in a queue and executes them when system load level permits. You end the <i>batch</i> command with the key combination <code>CTRL d</code> . |
| ps                      | outputs the status of the shell processes.                                                                                                                                      |
| kill PID                | terminates the shell process with the specified process ID ( <i>PID</i> ).                                                                                                      |
| nohup<br>command_list & | continues execution of the background process even if the user logs out.                                                                                                        |

Enabling the shell to access files

|                                                  |                                                                                                                                                                                                                                                                                                                                                                                                                      |
|--------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| chmod u+x <i>file_name</i>                       | grants the user execute permission for the file. This is particularly useful for shell script files.                                                                                                                                                                                                                                                                                                                 |
| mv <i>file_name</i> \$HOME/bin/ <i>file_name</i> | moves your file to the bin directory in your <i>home</i> directory. The bin directory contains executable shell scripts that you can access. Make sure that you include this directory in the <i>PATH</i> environment variable in your <i>.profile</i> file. The shell then searches in the <i>\$HOME/bin</i> directory for the file when you call it. Otherwise, the shell is unable to find and execute your file. |
| <i>file_name</i>                                 | The name of a file containing a shell script becomes a command name that you can use to call this shell script.                                                                                                                                                                                                                                                                                                      |

**Variables**

|                             |                                                                                                                                                                                                                               |
|-----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>positional_parameter</i> | A variable within a shell script that takes its value from an argument you specify in the command line of the program. Positional parameters consist of numbers preceded by a dollar sign (\$) ( <i>\$1, \$2, \$3 etc.</i> ). |
| echo                        | A command that allows you to display the value of a variable on your terminal.                                                                                                                                                |
| \$#                         | A special parameter that specifies the number of arguments used to call the shell script.                                                                                                                                     |
| \$*                         | A special parameter containing the values of all arguments used to call the shell script.                                                                                                                                     |
| <i>keyword_parameter</i>    | A variable to which the user can assign a value.                                                                                                                                                                              |

**Environment variables**

|         |                                                                                                                                                                          |
|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CDPATH  | defines the path for the <i>cd</i> command.                                                                                                                              |
| HOME    | Your <i>home</i> directory; the default variable for the <i>cd</i> command.                                                                                              |
| IFS     | The internal field separators (usually the blank, tab and carriage return).                                                                                              |
| LOGNAME | The user' s login name.                                                                                                                                                  |
| MAIL    | The name of the file you are using to store your electronic mail.                                                                                                        |
| PATH    | The path your login shell searches to find a command.                                                                                                                    |
| PS1     | The primary prompt. Default: (\$)                                                                                                                                        |
| PS2     | The secondary prompt. Default: (> ) The character defined in <i>PS2</i> appears at the start of a continuation line if the command you enter is more than one line long. |
| TERM    | The terminal type.                                                                                                                                                       |
| TZ      | Sets and stores the local time zone.                                                                                                                                     |



## 5.8 Summary of shell programming constructs

A here document has the following standard syntax:

```
command <<delimiter
...input lines ...
delimiter
```

A *for* loop has the following standard syntax:

```
for variable
 in list of values
do
 command1
 command2
 .
 .
 .
 last command
done
```

A *while* loop has the following standard syntax:

```
while
 command1
 command2
 .
 .
 .
 last command
do
 command1
 command2
 .
 .
 .
 last command
done
```

An *until* loop has the following standard syntax:

```
until
 command1
 command2
 .
 .
 .
 last command
do
 command1
 command2
 .
 .
 .
 last command
done
```

An *if...then* statement has the following standard syntax:

```
if
 command1
 command2
 .
 .
 .
 last command
then
 command2
 command2
 .
 .
 .
 last command
fi
```

An *if...then...else* statement has the following standard syntax:

```
if
 command1
 command2
 .
 .
 .
 last command
then
 command1
 command2
 .
 .
 .
 last command
else
 command1
 command2
 .
 .
 .
 last command
fi
```

A *case* statement has the following standard syntax:

```
case word
in
 pattern1)
 command line 1
 .
 .
 last command line
 ;;
 pattern2)
 command line 1
 .
 .
 last command line
 ;;
*)
 command line 1
 .
 .
 last command line
 ;;
esac
```

### **break and continue statements**

A *break* or *continue* statement causes the script to terminate any loop and go to the next command after the end of the loop.

---

## 6 The display editor vi

### 6.1 Introduction

*vi* (short for **visual**) is a display editor that enables you to create texts. Once you have become familiar with how to use it, you will find *vi* to be a valuable tool. The best way of learning how to use *vi* is to work through the examples and exercises given in this tutorial on your computer.

At the end of some of the sections, you will find references to exercises that will help you to put what you have learned into practice. The exercises themselves and their solutions are given at the end of this chapter.

Each section ends with a summary of the commands discussed. You will find a full summary of the *vi* commands, grouped together by function, and other information on *vi* in the *Commands* manual [5] under the heading *vi*.

Before you start to work with *vi*, you should have a basic knowledge of the Reliant UNIX file system. For further information, see chapter 3 *The file system*.

#### 6.1.1 What vi does

The *vi* editor is a powerful tool for creating and editing ASCII files. You control *vi* using only the alphanumeric keys on the keyboard; you do not need any of the function keys. This means that you can use any type of keyboard to control *vi*. The *vi* editor is available on all UNIX systems.

The *vi* editor can display one line or many lines of text. It allows you to move the cursor to any point on the screen or in the file and create, modify or delete text there. You can use other editor commands, such as global commands that allow you to replace several occurrences of the same string in a file by issuing one command. To move to another point in the file, you can scroll the text up or down. Scrolling displays the lines above or below the current window.

Not all terminals have text scrolling capability. This depends on the terminal type you are using. If in doubt, consult your system administrator.

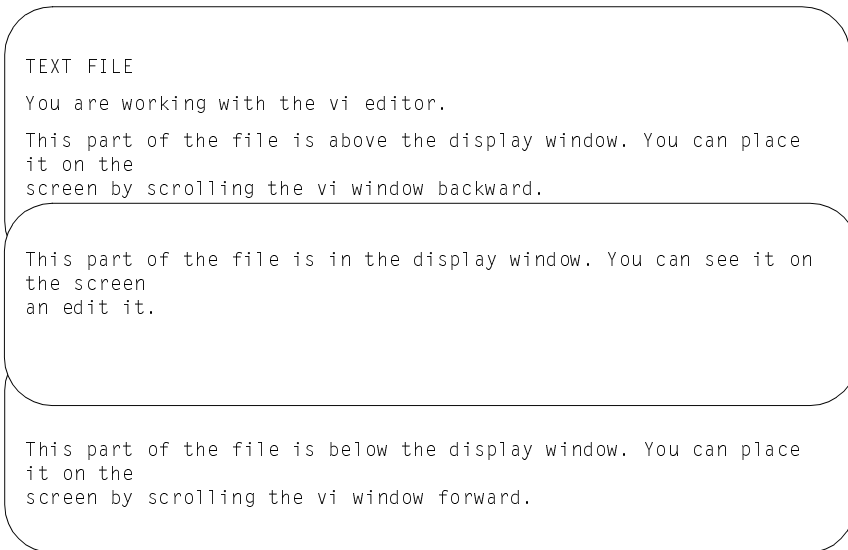


Figure 8: Displaying a file in a vi window

There are over 100 vi commands. This chapter covers the basic commands that will enable you to use vi simply but effectively. Specifically, it explains the following functions enabling you to

- call vi, create text, correct errors, save text in a file and quit the editor,
- move text within a file,
- cut and paste text,
- use special commands and shortcuts,
- use the line editor commands available in vi,
- temporarily escape to the shell to execute shell commands,
- recover a file that was lost when the editor session was interrupted,
- edit more than one file in the same session, and
- set the line wrap margin.

## 6.2 Getting started

In the following sections, you will learn how to call *vi* and set your working environment. You will also create your first file using the *vi* editor.

### 6.2.1 Setting your login environment

If you are going to use *vi* regularly, you should set up your login environment so that you do not have to configure your terminal each time you log in. If you specify the correct setting for your terminal configuration in the *.profile* file, your terminal will be configured automatically every time you log in. The section on *Modifying your login environment* in chapter 5 *The shell* tells you how to do this.

You use the *vi* command *set* to specify the default settings for the editor. If you want the default editor settings to be effective as soon as you call *vi*, you must enter the appropriate value of the *set* command in the *.exrc* file. You create this file in your *HOME* directory. The environment you set then remains in effect after you terminate your session. Information on all the available options is given under *ex* (all the commands of the *ex* editor are also available in *vi*) and *vi* in the *Commands* manual [5].

#### Calling *vi*

To start working with *vi*, you must call it by issuing a command to the shell:

```
$ vi
```

#### Displaying the *vi* options

First find out what default settings are in effect when *vi* is called. You use the following command under *vi* to check your settings for the terminal type and other options:

```
:set 
```

*vi* displays the current settings for the terminal type, line wrap margin and any other options that are set. You can also use the *:set* command to enter or modify various options.

## Setting the line wrap margin

By default, the line wrap margin is not set in *vi* and you have to press `↵` at the end of each line. If, however, you want the cursor to move to a new line automatically when it reaches a certain column, you can set the *wm* option (**w**rap**m**argin) using the *set* command.

First you must create a file called *.exrc* in your home directory, in which you enter options that define your *vi* editing environment. To create the *.exrc* file, call an editor with this file name. Then type in the line of text containing the specification for the line wrap margin (automatic line break) option. You use the following *vi* command to set this option:

```
:set wm=n
```

where *n* is the number of characters or columns from the right-hand side of the screen at which you want *vi* to break the text. This can happen only between words, however, and not between syllables within a word. When the cursor reaches the specified distance from the right-hand side of the screen, *vi* appends a newline automatically to the last word of each line entered. This control character is also assigned to `↵`.

The following example illustrates how to set the line wrap margin at a distance of five characters from the right-hand side of the screen using the *vi* editor:

Enter

```
$ cd
```

at shell level to go to your home directory, if this is not already your current directory.

Enter

```
$ vi .exrc
```

to call the *vi* editor and create a file called *.exrc* in your home directory. This file does not yet contain any entries.

Press `ESC` followed by a to prepare *vi* to accept your input.

Now enter the following text:

```
set wm=5
```

Press `ESC`. The text you can now see on the screen is stored in a buffer. You save the contents of this buffer and then quit the editor by pressing the following keys consecutively:

```
:wq
```



You have now set the line wrap margin and at the same time successfully completed your first *vi* session. The next time you call *vi*, the *.exrc* file will cause your default settings to be implemented.

### Setting the left margin

In input mode, *vi* indents each line in the same way as the previous line by inserting blanks and tabs. The start of the indented text depends how the line before is indented. If, for example, you insert tabs, the following lines are automatically indented with tabs.

You can use the *ai* option (autoindent) to set the left margin for your text. All the paragraphs then have the same indentation in relation to the left-hand edge of the screen.

If you want to set this option permanently, enter the following line in the *.exrc* file in your home directory:

```
set ai
```

### Displaying the operating mode

The *vi* editor operates in two modes: input mode, in which you can enter and replace text, and command mode, in which you can edit and modify text. When you start *vi*, it is automatically in command mode. For more detailed information on the operating modes, see the description of *Operating modes* in the section on *Creating a file*.

If you want *vi* to tell you when it is in input mode, you must include the entry *smd* (**showmode**) in the *.exrc* file in your home directory. To do this, enter the following line in this file:

```
set smd
```

## 6.3 Creating a file

First call the editor by entering the *vi* command with the name of the file you want to create or edit:

```
vi file_name
```

Let us assume that you want to create a file called *gazpacho*. Once you have entered the *vi* command with the file name *gazpacho*, *vi* clears the screen and displays a window in which you can enter and edit text. The *gazpacho* file is now opened:



Figure 9: The vi screen

The shaded box on the top line shows the position of the cursor (on the terminal, the cursor may be a blinking underscore or a reverse color block). Every other line is marked with a tilde (~), the symbol for an empty line. The status line at the bottom of the screen displays information. In this case, the message ""gazpacho" [New file]" tells you that you have just created a file called *gazpacho*.

### 6.3.1 Operating modes

The *vi* editor operates in two modes:

- input mode, in which you can
  - enter text
  - replace text
  - overwrite text
- command mode, in which you can
  - edit and modify existing text
  - delete, move and copy text
  - search for text
  - move to another point in the file

*vi* automatically starts in command mode. Many *vi* commands consist of only one or two characters. This means that, once you have switched to command mode, *vi* interprets the next character you enter as command information. These commands take effect immediately; you do not have to press `[↵]` to initiate them.

If you enter a command used to enter or extend text, *vi* switches to input mode. This means that *vi* interprets any characters you enter as text characters. To return to command mode, press `[ESC]`.

The keyboard therefore has a dual function. You can use it both to enter your text and as a control device for entering commands. This allows you to modify your text quickly using simple command sequences, although you must always be aware of the mode in which *vi* is currently working.

#### Caution:

If you are working with *vi* in command mode and you want to enter text, you may accidentally issue a command that has unexpected results. If you are not sure whether *vi* is in command or input mode, simply press `[ESC]` to make sure that the editor has returned to command mode. It does not do any harm to press `[ESC]` too often.

6.3.2 Creating text in input mode

Once you have called *vi*, you are in command mode and *vi* is waiting for your commands. You issue commands by typing in the appropriate character sequences

Entering new text

There are two basic commands for entering text in an existing file:

|   |             |
|---|-------------|
| i | Insert text |
| a | Append text |

Table 52: Entering new text

If you want to insert text in a section of text that is already displayed on the screen, first use the commands *h*, *j*, *k* and *l* to move the cursor to the desired position. Then press *i* and start to enter your text. The new characters you enter are inserted to the left of the current cursor position. The character on which the cursor is currently positioned and all the characters that follow it are displaced to the right to make room for the new text. *vi* continues to process the characters you enter in this way until you press ESC. If necessary, *vi* moves the original characters onto the next line.

The append command *a* has the same function, except that the characters are inserted in the text to the right of the current cursor position.

You can also move the cursor to different positions on the screen, or page through a file to insert or delete characters, words or lines. Further information on these functions is given in the sections of this chapter that describe the cursor positioning commands, in particular the section on *Positioning the cursor*.

Enter the *a* (append) command. The *a* command does not appear on the screen. Then press ↵ to switch from command mode to input mode. You can now insert text in the file.

Enter some text. If you want to move to a new line, press ↵.

If, however, you have set the line wrap margin in the *.exrc* file, *vi* automatically breaks the text as soon as the current line is full and you can continue typing on the new line (see *Setting the line wrap margin* in section 6.2.1 *Setting your login environment*).

If you hear an audible signal after pressing `[ESC]`, you are already in command mode. Pressing `[ESC]` in command mode has no effect on the text in the file, even if you do it several times. If you are in doubt, you can check the current operating mode of the editor in this way.

**Exiting input mode**

Once you have finished entering text, press `[ESC]` to exit input mode and return to command mode. You can now edit the text you have just created or store it in a file.

# 6.4 Creating text

There are three basic ways of creating text. You can append new text to existing text using the `a` (**a**ppend) command. Alternatively, you can insert new text into existing text using the `i` (**i**nsert) command. Finally, you can use the `o` (**o**pen) command to create a blank line in which you can enter new text.

Once you have finished entering text using one of these commands, you return to *vi* command mode by pressing `[ESC]`.

## 6.4.1 Appending text

|                |                                                         |
|----------------|---------------------------------------------------------|
| <code>a</code> | Append text to the right of the current cursor position |
| <code>A</code> | Append text at the end of the current line.             |

Table 53: Appending text

You have already learned about the `a` command in section 6.3.2 *Creating text in input mode*. Create a new file called *test1* and use the `a` command to enter text in it. Then press `[ESC]` to return to *vi* command mode. Now try using the `A` command and compare the results.

### 6.4.2 Inserting text

|   |                                                                                           |
|---|-------------------------------------------------------------------------------------------|
| i | Insert text to the left of the current cursor position.                                   |
| I | Insert text at the start of the current line to the left of the first nonblank character. |

Table 54: Inserting text

Use the `i` command to enter text in the `test1` file. Then press `[ESC]` to return to command mode. Now try using the `I` command and compare the results. Once you have finished, press `[ESC]` to return to command mode.

The following examples illustrate the different functions of the `a` and `i` commands. The shaded box indicates the current cursor position, where you can enter new text. Because the commands you enter do not appear on the screen, the examples make two basic assumptions:

- You have set the `smd` (*showmode*) option. The mode that corresponds to the command you are using is then displayed on the status line.
- Once you have made changes, you save the contents of the editor buffer using the `w` command. This causes the current number of lines and characters to be displayed on the status line. For a more detailed description of this command, see section 6.13 *Line editor commands*.

To save space, only the first lines of text and the status line are shown in the following examples. The missing middle section of the screen is indicated by dots.

#### Inserting text at the start of a line - the `I` command

Ingredients:

ne

cucumber, peeled, seede and dice

.

.

.

"gazpacho" 3 lines, 42 characters

INSERT MODE

Once you have inserted the O at the start of the first line of text, press **[ESC]** to leave input mode. Then position the cursor in front of the word *cucumber*. To insert the word *large* at this position, you must return to input mode, this time using **i**:

### Inserting text within a line - the i command

Ingredients:

One\_cucumber, peeled, seede and dice

.  
.  
.

"gazpacho" 3 lines, 43 characters

INSERT MODE

You now want to add the missing letter to the word *seede*. Exit input mode once again using **[ESC]**, position the cursor on the final character of *seede* and return to input mode using the **a** command. This command automatically moves the cursor to the next column:

### Appending text to a character - the a command

Ingredients:

One large cucumber, peeled, seede\_ and dice

.  
.  
.

"gazpacho" 3 lines, 48 characters

APPEND MODE

Finally, you want to append a d at the end of the second line of text. Press **[ESC]** to exit input mode and enter the **A** command to append text at the end of the line. This command, too, automatically moves the cursor one column to the right:

Appending text at the end of a line - the A command

Ingredients:  
One large cucumber, peeled, seeded and dice\_  
.   
.   
.   
"gazpacho" 3 lines, 49 characters

APPEND MODE

Exit input mode once again by pressing `[ESC]`. This moves the cursor back by one character, positioning it on the final letter *d*:

Ingredients:  
One large cucumber, peeled, seeded and diced\_  
.   
.   
.   
"gazpacho" 3 lines, 50 characters

6.4.3 Inserting a blank line

|   |                                                                                                                                                 |
|---|-------------------------------------------------------------------------------------------------------------------------------------------------|
| o | Insert a blank line below the current line. You can enter text on this new line. You can enter this command anywhere on the current line.       |
| O | Insert a blank line above the current line. You can enter text on this new line. You can enter this command, too, anywhere on the current line. |

Table 55: Inserting a blank line

You use the `O` and `o` commands to insert a blank line above or below the current line. At the same time, these commands cause *vi* to switch to input mode.

The following examples illustrate the different functions of the `O` and `o` commands. These examples (like the examples for the `a` and `i` commands) assume that you have set the *smd* (*showmode*) option and that you save any changes using `w`.



## Inserting a blank line above the current line - the O command

```
_Ingredients:
One large cucumber, peeled, seeded and diced
.
.
.
""gazpacho" 3 lines, 50 characters
```

The cursor is currently located in the first column of the first line of text. To insert a blank line above this line, enter an uppercase `O`. `O` switches to input mode and moves the cursor to the first column of the new line. You can now enter text on this line, e.g. *Gazpacho*:

```
Gazpacho_
Ingredients:
One large cucumber, peeled, seeded and diced
.
.
.
""gazpacho" 4 lines, 58 characters
```

OPEN MODE

If you want to insert more blank lines, simply press `↵`. Each time you enter a newline character in *OPEN MODE*, a new blank line is created automatically. If, however, you have already exited input mode, you can insert more lines below the new blank line using `o`. `o` then behaves in the same way as `O`:

### Inserting a blank line below the current line - the o command

```
Gazpacho
(a spanish soup)_
Ingredients:
```

```
One large cucumber, peeled, seeded and diced
.
.
.
```

```
"gazpacho" 5 lines, 72 characters
```

OPEN MODE

To make the text easier to read, use **[↓]** to insert another blank line between the subtitle and the third line of text. Then press **[ESC]** to leave input mode. Your screen will look like this:

```
Gazpacho
(a spanish soup)
_
Ingredients:
```

```
One large cucumber, peeled, seeded and diced
.
.
.
```

```
"gazpacho" 6 lines, 73 characters
```

OPEN MODE

Table 56 summarizes the *vi* commands that you can use to create a file and add text.

| Command          | Function                                                                  |
|------------------|---------------------------------------------------------------------------|
| a                | Append text to the right of the current cursor position.                  |
| A                | Append text at the end of the current line.                               |
| i                | Insert text to the left of the current cursor position.                   |
| I                | Insert text in front of the first nonblank character in the current line. |
| o                | Enter text at the start of a new line inserted below the current line.    |
| O                | Enter text at the start of a new line inserted above the current line.    |
| <span>ESC</span> | Exit one of the above input modes and return to <i>vi</i> command mode.   |

Table 56: Summary of the commands for creating text

Now try Exercise 1 at the end of this chapter.

## 6.5 Editing text in command mode

This section shows you how to display certain parts of a text in order to edit them.

You cannot edit an existing file unless the section you want to edit is displayed on the screen. Editing an existing file involves inserting, modifying and deleting text. *vi* provides you with several options for paging through a file and moving the cursor to a different line or a different position on the current line.

### 6.5.1 Positioning the cursor

If you want to edit a particular section of text, you must move the cursor to the corresponding position on the screen. You can do this easily using four keys that are grouped together on the keyboard: h, j, k, and l.

|    |                                                   |
|----|---------------------------------------------------|
| h  | Move the cursor one character to the left.        |
| nh | Move the cursor <i>n</i> characters to the left.  |
| j  | Move the cursor down one line.                    |
| nj | Move the cursor down <i>n</i> lines.              |
| k  | Move the cursor up one line.                      |
| nk | Move the cursor up <i>n</i> lines.                |
| l  | Move the cursor one character to the right.       |
| nl | Move the cursor <i>n</i> characters to the right. |

Table 57: Positioning the cursor

The j and k commands keep the cursor in the same column. If, for example, the cursor is in column 7, it moves to the seventh column of the new line when you type j or k. If the new line contains only five characters, for example, the cursor moves to the last character on the line.

If you want to move the cursor several characters to the left or right or several lines up or down, you do not have to enter these command keys repeatedly. Simply precede the command with the desired number of characters or lines. If you want to move the cursor two characters to the right, you can either press l twice or enter 2l. To move the cursor up four lines, either press k four times or enter 4k. If it is no longer possible to move the cursor in the direction you have specified, vi issues an audible signal.

Some terminals, such as the 97801, have extended keyboards featuring an additional keypad with special arrow keys for moving the cursor. These keys provide the same functionality as the h, j, k and l commands:

|                                                                                     |                                             |
|-------------------------------------------------------------------------------------|---------------------------------------------|
|  | Move the cursor one character to the left.  |
|  | Move the cursor down one line.              |
|  | Move the cursor up one line.                |
|  | Move the cursor one character to the right. |

Table 58: Moving the cursor using the arrow keys

The following example illustrates how the above commands work:

Ingredients:

```
One large cucumber, peeled, seeded and diced
A small onion, chopped
Three tomatoes, peeled and diced
One small green pepper, seeded and diced
Four tablespoons olive oil
Garlic to taste
Lemon juice
~
~
~
```

Try to move the cursor to a different position using the `j` and `k` commands. First move it up eight lines:

`[ESC] 8k`

The cursor moves up eight lines above the current line. If, for example, there are only five lines above the current line, an audible signal is issued and the cursor remains on the current line.

Now move the cursor down five lines by entering

`[ESC] 5j`

`vi` clears and redraws the screen. The cursor is now in the middle of the new screen. The current line is now five lines below the line on which the cursor was originally positioned. If, for example, there are only three lines below the current line, an audible signal is issued and the cursor remains on the current line. Enter the following command and see what happens:

`[ESC] 35k`

An audible signal is issued and the cursor remains on the current line. You cannot move the cursor up since the file contains less than 35 lines.

The cursor positioning commands `h`, `j`, `k` and `l` do not appear on the screen when you enter them. Characters are only displayed on the screen if the editor is in input mode and you are adding text to your file.

If you try to enter a positioning command and the command letters appear on the screen, the editor is still in input mode. If this happens, press `[ESC]` to switch to command mode and try entering the commands again.

In addition to the positioning commands `h` and `l`, you can use the space bar `[ ]` and the backspace key `[␣]` to move the cursor to a character to the left or right of the current cursor position on the current line.

If you enter a number before pressing a command key, the cursor will move the specified number of characters to the left or right.

|                                                                                            |                                                   |
|--------------------------------------------------------------------------------------------|---------------------------------------------------|
|           | Move the cursor one character to the left.        |
| <i>n</i>  | Move the cursor <i>n</i> characters to the left.  |
|           | Move the cursor one character to the right.       |
| <i>n</i>  | Move the cursor <i>n</i> characters to the right. |

Table 59: Summary of the positioning commands

The following example illustrates how to create a table and correct typing mistakes. Let us assume that you want to create a file containing several columns. Once you have entered the headings for the first three columns, you notice a typing mistake:

```
Column 1 Column 2 column_
~
~
~
```

You want to correct your typing mistake now, before entering any more characters. Exit insert mode and press **ESC** to return to command mode. The cursor moves one character to the left and is now positioned on the letter n:

```
Column 1 Column 2 column
~
~
~
```

Now move the cursor five characters to the left by pressing **h** five times.

```
Column 1 Column 2 _column
~
~
~
```

Delete the letter `c` by pressing `x` (short for `extinguish`). This letter disappears. Now switch to input mode by pressing `i` (short for `insert`). Type in the letter `C` and press `[ESC]`.

```
Column 1 Column 2 Column
~
~
~
```

You can now move the cursor back to its original position at the end of the line by entering `5l`.

```
Column 1 Column 2 Column
~
~
~
```

If you want to enter more text at the end of the file, use the `a` command. To insert text at a particular point within the text, use the `i` command.

### 6.5.2 Deleting text

To delete a character, position the cursor on it and press `x` (short for `extinguish`). Watch what happens on the screen: the character disappears from the screen and the following characters all move one character to the left. You can delete three consecutive characters, for example, by pressing `x` three times. If you precede the `x` command with a number  $n$ , you delete  $n$  characters to the right of and including the character on which the cursor is positioned.

**Note:**

After you press `[ESC]`, the cursor may move one character to the left. It is better to press `[ESC]` twice to be certain that you are in the correct mode. Always check the current position of the cursor to avoid making unintentional changes to your text.

|                              |                                                                                  |
|------------------------------|----------------------------------------------------------------------------------|
| <code>x</code>               | Delete the character on which the cursor is positioned.                          |
| <code>n<math>x</math></code> | Delete $n$ characters to the right of and including the current cursor position. |

Table 60: Deleting a character

Let us assume, for example, that you want to correct the recipe in the *gazpacho* file:

Ingredients:

```
One large cucumber, peeled, seeded and diced
A small onion, chopped
Three tomatoes, peeled and diced
One small green pepper, seeded and diced
Four tablespoons finest sunflower or olive oil
Garlic to taste
Lemon juice
~
~
~
```

To delete the character on which the cursor is positioned, as well as the characters that follow it, enter `x` with the number of characters you want to delete. If, for example, you want to delete the word *finest* and the blank that follows it, move the cursor onto the first letter of this word and delete seven characters:

`[ESC]` 7x

Ingredients:

```
One large cucumber, peeled, seeded and diced
A small onion, chopped
Three tomatoes, peeled and diced
One small green pepper, seeded and diced
Four tablespoons sunflower or olive oil
Garlic to taste
Lemon juice
~
~
~
```

Note that the characters to the right of the word you have deleted automatically move to the left to close up the gap in the text.

Now make the text even shorter: since Gazpacho does not taste authentic unless you use olive oil, you want to delete the words *sunflower* and *or*. Delete 13 characters (the 9 letters in the word *sunflower*, the 2 letters in the word *or*, the blank between the two words and the blank that follows *or*. Enter:

`[ESC]` 12x



```
Ingredients:
One large cucumber, peeled, seeded and diced
A small onion, chopped
Three tomatoes, peeled and diced
One small green pepper, seeded and diced
Four tablespoons olive oil
Garlic to taste
Lemon juice
~
~
~
```

If the string you want to delete is a whole word, you can use a special command provided by vi: `[ESC] dw` (short for **d**ele**e**te **w**ord). For further information on this command, see section 6.7.3 *Word-oriented positioning*.

## 6.6 Quitting vi

The text you have entered so far is stored in a temporary buffer. Once you have finished entering text, you can write the buffer contents to a file and return to the shell.

To do this, you press `Z` twice. The editor saves the file name you specified at the start of the editing session when you called `vi` from the shell. The text you have stored in the buffer is written to this file. A message at the bottom of the screen tells you the file name and size (number of lines and characters in the file). The shell prompt then appears.

The following example illustrates the above procedure:

Ingredients:

```
One large cucumber, peeled, seeded and diced
A small onion, chopped
Three tomatoes, peeled and diced
One small green pepper, seeded and diced
Four tablespoons olive oil
Garlic to taste
Lemon juice

~
~
~
~
~
~
~
~
~
~
~
~
```

"gazpacho" [New file] 9 lines, 172 characters

You quit the editor by entering

ESC ZZ  
\$

Instead of the ZZ command, you could use the command :wq  to achieve the same results. The :w  command (**w**rite) saves the buffer contents to a file. You use the :q  command (**q**uit) to exit the editor and return to the shell.

You can specify these two commands either on two separate lines:

: w 

: q 

or consecutively on a single line:

:wq 

These commands are known as line editor commands. They begin with a colon and appear on the bottom line of the screen.

### 6.6.1 Summary

Table 61 summarizes the basic commands of the *vi* editor.

| Command                                                                                        | Function                                                             |
|------------------------------------------------------------------------------------------------|----------------------------------------------------------------------|
| <code>vi file_name</code>                                                                      | Call the <i>vi</i> editor to edit the file called <i>file_name</i> . |
| <code>a</code>                                                                                 | Append text to the right of the cursor.                              |
| <code>h</code>                                                                                 | Move the cursor one character to the left.                           |
| <code>j</code>                                                                                 | Move the cursor down one line.                                       |
| <code>k</code>                                                                                 | Move the cursor up one line.                                         |
| <code>l</code>                                                                                 | Move the cursor one character to the right.                          |
| <code>x</code>                                                                                 | Delete a character.                                                  |
| <code></code> | Return key (carriage return).                                        |
| <code></code> | Exit input mode and return to command mode.                          |
| <code>:w</code>                                                                                | Write the buffer contents to a file.                                 |
| <code>:q</code>                                                                                | Quit <i>vi</i> .                                                     |
| <code>:wq</code>                                                                               | Write the buffer contents to a file and quit <i>vi</i> .             |
| <code>ZZ</code>                                                                                | Write the buffer contents to a file and quit <i>vi</i> .             |

Table 61: Summary of basic commands

Now try Exercise 2 at the end of this chapter.

## 6.7 Positioning the cursor

So far you have used the `h`, `j`, `k` and `l` commands, `` and the space bar `` to move the cursor in the text. However, *vi* provides you with several other commands that enable you to move the cursor quickly around the screen. This section describes:

- character-oriented positioning on a line
- line-oriented positioning

- positioning by text objects
  - words
  - sentences
  - paragraphs
- screen-oriented positioning

There are also commands that position the cursor within parts of the *vi* editing buffer that are not displayed on the screen. These commands are described in section 6.8 *Positioning the cursor in undisplayed text*, which follows this section.

To follow this section, you should call *vi* with a file that contains at least 40 lines. If you do not have a file of that length, first create one. Always remember that you must be in *vi* command mode in order to enter the commands described here. If in doubt, press `[ESC]` to make absolutely sure that you are not in input mode.

6.7.1 Positioning the cursor on a character

There are three ways of positioning the cursor on another character on the same line:

- Move the cursor one character to the left or right.
- Specify a character to move the cursor to the start or end of the line.
- Search for a character in a line.

You are already familiar with the first method. The following sections explain the other two methods.

Moving the cursor to the start or end of a line

There are three commands that allow you to move the cursor to the first or last character in a line.

|          |                                                            |
|----------|------------------------------------------------------------|
| \$       | Move the cursor to the last character in a line.           |
| 0 (zero) | Move the cursor to the first character in a line.          |
| ^        | Move the cursor to the first nonblank character in a line. |

Table 62: Moving the cursor to the start or end of a line

The following examples show how these three commands affect the cursor position.

Curries

Curries originate from many different parts of the world. However, the term is chiefly associated with Asian cooking. The special flavor of each curry depends on the careful selection and blending of the individual spices.

.

For example, you use cloves, cardamoms, cinnamon, bay leaves, and black pepper to make a warm combination of spices known as "Garam Masala". You can add chillies to make a hot curry that will bring tears to your eyes.

.

Even a very simple Asian meal will consist of several dishes and a wide range of side dishes and pickles, all served at the same time.

.

.

.

"curries" 12 lines 483 characters

You move the cursor to the end of the line using \$:

Curries

Curries originate from many different parts of the world. However, the term is chiefly associated with Asian cooking. The special flavor of each curry depends on the careful selection and blending of the individual spices.

.

For example, you use cloves, cardamoms, cinnamon, bay leaves, and black pepper to make a warm combination of spices known as "Garam Masala". You can add chillies to make a hot curry that will bring tears to your eyes.

.

Even a very simple Asian meal will consist of several dishes and a wide range of side dishes and pickles, all served at the same time.

.

.

.

"curries" 12 lines 483 characters

You move the cursor to the start of the line using 0:

```
Curries
Curries originate from many different parts of the world. However, the term is
chiefly associated with Asian cooking. The special flavor of each curry
depends on the careful selection and blending of the individual spices.
.
For example, you use cloves, cardamoms, cinnamon, bay leaves, and black
pepper to make a warm combination of spices known as "Garam Masala". You
can add chillies to make a hot curry that will bring tears to your eyes.
.
Even a very simple Asian meal will consist of several dishes an a
wide range of side dishes and pickles, all served at the same time.
.
.
.
"curries" 12 lines 483 characters
```

You move the cursor to the first nonblank character in the line by entering a caret ^:

```
Curries
Curries originate from many different parts of the world. However, the term is
chiefly associated with Asian cooking. The special flavor of each curry
depends on the careful selection and blending of the individual spices.
.
For example, you use cloves, cardamoms, cinnamon, bay leaves, and black
pepper to make a warm combination of spices known as "Garam Masala". You
can add chillies to make a hot curry that will bring tears to your eyes.
.
Even a very simple Asian meal will consist of several dishes an a
wide range of side dishes and pickles, all served at the same time.
.
.
.
"curries" 12 lines 483 characters
```

```
Curries

Curries originate from many different parts of the world. However, the term is
chiefly associated with Asian cooking. The special flavor of each curry
depends on the careful selection and blending of the individual spices.
.
For example, you use cloves, cardamoms, cinnamon, bay leaves, and black
pepper to make a warm combination of spices known as "Garam Masala". You
can add chillies to make a hot curry that will bring tears to your eyes.
.
Even a very simple Asian meal will consist of several dishes an a
wide range of side dishes and pickles, all served at the same time.
.
.
.
"curries" 12 lines 483 characters
```

Searching for a character in a line

The third way of positioning the cursor in the current line is to search for a specific character on this line. If the character is not found, an audible signal is issued and the cursor position remains unchanged. You can use the following six commands to search within a line: f, F, t, T, (;) and (,). The f, F, t, and T commands must be followed by a character; you do not have to specify a character after the semicolon (;) and comma (,) commands.

|                |                                                                                                     |
|----------------|-----------------------------------------------------------------------------------------------------|
| f <sub>x</sub> | Move the cursor to the right and position it on <i>x</i> .                                          |
| F <sub>x</sub> | Move the cursor to the left and position it on <i>x</i> .                                           |
| t <sub>x</sub> | Move the cursor to the right, positioning it on the character immediately to the left of <i>x</i> . |
| T <sub>x</sub> | Move the cursor to the left, positioning it on the character immediately to the right of <i>x</i> . |
| ;              | Continue the search specified in the last command, in the same direction.                           |
| ,              | Continue the search specified in the last command, but in the opposite direction.                   |

Table 63: Searching for strings

In the next example, *vi* searches the current line to the right for the first occurrence of the letter *a*:

```
Curries

Curries originate from many different parts of the world. However, the term is
chiefly associated with Asian cooking. The special flavor of each curry
depends on the careful selection and blending of the individual spices.
.
For example, you use cloves, cardamoms, cinnamon, bay leaves, and black
pepper to make a warm combination of spices known as "Garam Masala". You
can add chillies to make a hot curry that will bring tears to your eyes.
.
Even a very simple Asian meal will consist of several dishes an a
wide range of side dishes and pickles, all served at the same time.
.
.
.
"curries" 12 lines 483 characters
```

```
Curries

Curries originate from many different parts of the world. However, the term is
chiefly associated with Asian cooking. The special flavor of each curry
depends on the careful selection and blending of the individual spices.
.
For example, you use cloves, cardamoms, cinnamon, bay leaves, and black
pepper to make a warm combination of spices known as "Garam Masala". You
can add chillies to make a hot curry that will bring tears to your eyes.
.
Even a very simple Asian meal will consist of several dishes an a
wide range of side dishes and pickles, all served at the same time.
.
.
.
"curries" 12 lines 483 characters
```



### 6.7.2 Line-oriented positioning

You have already used the `j` and `k` commands to move the cursor to a different line. The following commands are also available:

|                                                                                   |                                                                                            |
|-----------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|
| -                                                                                 | Move the cursor one line up, positioning it on the first nonblank character in the line.   |
|  | Move the cursor one line down, positioning it on the first nonblank character in the line. |
| +                                                                                 | Move the cursor one line down, positioning it on the first nonblank character in the line. |

Table 64: Line-oriented positioning

#### The minus sign positioning command (-)

The `-` (minus) command moves the cursor up one line, positioning it on the first nonblank character, if any, on the line. If you want to move the cursor more than one line at a time, precede the `-` command with the number of lines you want the cursor to move. For example, you enter the following command to move the cursor up 13 lines:

13-

If some of these lines are above the current window, the window will scroll backward to reveal them.

If you attempt to move the cursor up 100 lines by entering

100-

and there are only 80 lines above the current line, the cursor position remains unchanged and an audible signal is issued.

#### The plus sign positioning command (+)

The `+` or  command moves the cursor to the first nonblank character on the next line. You can use the `+` command to move the cursor several lines at a time. If, for example, you want to move the cursor down nine lines, enter

9+

If some of these lines are below the current window, the window will scroll forward to reveal them.

The  positioning command

The  command has the same effect as the + command.

6.7.3 Word-oriented positioning

vi considers a word to be a string of characters that may include letters, numbers and underscores.

You can use the following six commands to move the cursor word by word: w, b, e, W, B and E.

The lowercase commands (w, b and e) treat any character other than a letter, number or underscore as a delimiter. Delimiters signify the beginning or end of a word. Punctuation before or after a blank is also interpreted as a word. The beginning or end of a line is also a delimiter.

The uppercase commands (W, B and E) interpret punctuation as part of the word. Words are delimited by white space (blanks, tabs or newlines).

The functions of the individual word-oriented positioning commands are described below

|    |                                                                                                                                                                                                                                                                                |
|----|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| w  | Move the cursor to the first character of the next word. If you want to move the cursor several words at a time, you can either press w several times or precede the w command with the desired number of words.                                                               |
| nw | Move the cursor to the first character of the <i>n</i> th word to the right of the current cursor position. If this word is not on the current line, the cursor automatically moves to the relevant line and continues to count the words from the beginning of the next line. |

Table 65: Using w to move the cursor word by word

To move the cursor to the start of the tenth word to the right of the current word, enter:

10w

```
Curries

Curries originate from many different parts of the world. However, the term is
chiefly associated with Asian cooking. The special flavor of each curry
depends on the careful selection and blending of the individual spices.
.
For example, you use cloves, cardamoms, cinnamon, bay leaves, and black
pepper to make a warm combination of spices known as "Garam Masala". You
can add chillies to make a hot curry that will bring tears to your eyes.
.
Even a very simple Asian meal will consist of several dishes an a
wide range of side dishes and pickles, all served at the same time.
.
.
.
"curries" 12 lines 483 characters
```

```
Curries

Curries originate from many different parts of the world. However, the term is
chiefly associated with Asian cooking. The special flavor of each curry
depends on the careful selection and blending of the individual spices.
.
For example, you use cloves, cardamoms, cinnamon, bay leaves, and black
pepper to make a warm combination of spices known as "Garam Masala". You
can add chillies to make a hot curry that will bring tears to your eyes.
.
Even a very simple Asian meal will consist of several dishes an a
wide range of side dishes and pickles, all served at the same time.
.
.
.
"curries" 12 lines 483 characters
```

|   |                                                                                    |
|---|------------------------------------------------------------------------------------|
| W | Move the cursor to the word that follows the next blank, ignoring all punctuation. |
| e | Move the cursor to the last character of the next word.                            |

Table 66: Using W and e to move the cursor word by word

If you want to move the cursor from the current word to the word that follows the next blank, enter: W

Curries

Curries originate from many different parts of the world. However, the term is chiefly associated with Asian cooking. The special flavor of each curry depends on the careful selection and blending of the individual spices.

.

For example, you use cloves, cardamoms, cinnamon, bay leaves, and black pepper to make a warm combination of spices known as "Garam Masala". You can add chillies to make a hot curry that will bring tears to your eyes.

.

Even a very simple Asian meal will consist of several dishes an a wide range of side dishes and pickles, all served at the same time.

.

.

.

"curries" 12 lines 483 characters

Curries

Curries originate from many different parts of the world. However, the term is chiefly associated with Asian cooking. The special flavor of each curry depends on the careful selection and blending of the individual spices.

.

For example, you use cloves, cardamoms, cinnamon, bay leaves, and black pepper to make a warm combination of spices known as "Garam Masala". You can add chillies to make a hot curry that will bring tears to your eyes.

.

Even a very simple Asian meal will consist of several dishes an a wide range of side dishes and pickles, all served at the same time.

.

.

.

"curries" 12 lines 483 characters

|    |                                                                                                                                                                                                                                             |
|----|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| E  | Ignore all punctuation except blanks. Only white space is interpreted as a delimiter.                                                                                                                                                       |
| b  | Move the cursor backward on the line to the first character of the previous word.                                                                                                                                                           |
| nb | Move the cursor backward by $n$ words to the first character of the $n$ th word. The b command does not stop at the beginning of the line, but moves the cursor to the end of the line above and continues moving it backward on this line. |
| B  | Can be used just like the b command, except that it considers only blanks and newlines to be delimiters. It treats all other punctuation as letters of a word.                                                                              |

Table 67: Using E, b and B to move the cursor word by word

If you want to move the cursor backward by three words and position it on the first character of the third word, enter:

3b

```
Curries

Curries originate from many different parts of the world. However, the term is
chiefly associated with Asian cooking. The special flavor of each curry
depends on the careful selection and blending of the individual spices.
.
For example, you use cloves, cardamoms, cinnamon, bay leaves, and black
pepper to make a warm combination of spices known as "Garam Masala". You
can add chillies to make a hot curry that will bring tears to your eyes.
.
Even a very simple Asian meal will consist of several dishes an a
wide range of side dishes and pickles, all served at the same time.
.
.
.
"curries" 12 lines 483 characters
```

```
Curries

Curries originate from many different parts of the world. However, the term is
chiefly associated with Asian cooking. The special flavor of each curry
depends on the careful selection and blending of the individual spices.
.
For example, you use cloves, cardamoms, cinnamon, bay leaves, and black
pepper to make a warm combination of spices known as "Garam Masala". You
can add chillies to make a hot curry that will bring tears to your eyes.
.
Even a very simple Asian meal will consist of several dishes an a
wide range of side dishes and pickles, all served at the same time.
.
.
.
"curries" 12 lines 483 characters
```

### 6.7.4 Sentence-oriented positioning

The *vi* editor also recognizes sentences. *vi* assumes that a sentence ends in an exclamation mark (!), a period (.) or a question mark (?). If these delimiters appear in the middle of a line, they must be followed by two blanks otherwise *vi* does not recognize them as the end of a sentence. You should make a habit of entering two blanks after a period (.) at the end of each sentence, since this enables you to manipulate sentences as units.

You can move the cursor sentence by sentence using the left parenthesis ( ( ) and right parenthesis ( ) ) commands

|    |                                                                                                |
|----|------------------------------------------------------------------------------------------------|
| (  | Move the cursor to the start of the current sentence.                                          |
| n( | Move the cursor to the start of the <i>n</i> th sentence to the left of the current sentence.  |
| )  | Move the cursor to the start of the next sentence.                                             |
| n) | Move the cursor to the start of the <i>n</i> th sentence to the right of the current sentence. |

Table 68: Sentence-oriented positioning

The following example illustrates how to move the cursor around the screen using the left parenthesis command.

To move the cursor to the start of the current sentence, enter

(

Word soup

Word word. Word word word word word word word word word word word  
word word word. Word word word word word word word word word word  
word word word word. Word word word word word word word word word  
word word word word word. Word word word word word word word word  
word word word word word word. Word word word word word word word  
word word word word word word. Word word word word word word.

Word word word word word word word word. Word word word word word  
word word word word word word word word word. Word word word word  
word word word word word word word word word word. Word word word  
word word word word word word word word word word word. Word  
word word word word word word word word word word word word.

.  
.   
.

"soup" 14 lines, 811 characters

Word soup

Word word. Word word word word word word word word word word word  
word word word. Word word word word word word word word word word  
word word word word. Word word word word word word word word word  
word word word word word. Word word word word word word word word  
word word word word word word. Word word word word word word word  
word word word word word word. Word word word word word word.

Word word word word word word word word. Word word word word word  
word word word word word word word word word. Word word word word  
word word word word word word word word word word. Word word word  
word word word word word word word word word word word. Word  
word word word word word word word word word word word word.

.  
.   
.

"soup" 14 lines, 811 characters



### 6.7.5 Paragraph-oriented positioning

`vi` interprets text preceded by a blank line as a paragraph. If you want to be able to move the cursor to the start of a paragraph (or delete or overwrite an entire paragraph), make sure you insert a blank line at the end of each paragraph.

To move the cursor paragraph by paragraph, you use the left brace (`{`) and right brace (`}`) commands

|                 |                                                                                        |
|-----------------|----------------------------------------------------------------------------------------|
| <code>{</code>  | Move the cursor to the start of the current paragraph.                                 |
| <code>n{</code> | Move the cursor to the start of the <i>n</i> th paragraph above the current paragraph. |
| <code>}</code>  | Move the cursor to the start of the next paragraph.                                    |
| <code>n}</code> | Move the cursor to the <i>n</i> th paragraph below the current line.                   |

Table 69: Paragraph-oriented positioning

The following examples show you how to move the cursor to the start of a different paragraph. To move the cursor to the start of the next paragraph, enter:

```
}

Word soup

Word word. Word word word word word word word word word word word
word word word. Word word word word word word word word word word
word word word word. Word word word word word word word word word
word word word word word. Word word word word word word word word
word word word word word word. Word word word word word word word
word word word word word word. Word word word word word word.

Word word word word word word word word. Word word word word word
word word word word word word word word word. Word word word word
word word word word word word word word word word. Word word word
word word word word word word word word word word word. Word word
.
.
.

"soup" 14 lines, 811 characters
```

```
Word soup

Word word. Word word word word word word word word word word word
word word word. Word word word word word word word word word word
word word word word. Word word word word word word word word word
word word word word word. Word word word word word word word word
word word word word word word. Word word word word word word word
word word word word word word. Word word word word word word.

Word word word word word word word word. Word word word word word
word word word word word word word word word. Word word word word
word word word word word word word word word word. Word word word
word word word word word word word word word word word. Word
.
.
.

"soup" 14 lines, 811 characters
```

6.7.6 Screen-oriented positioning

The vi editor provides you with three commands for positioning the cursor on the screen

|   |                                                   |
|---|---------------------------------------------------|
| H | Move the cursor to the first line on the screen.  |
| M | Move the cursor to the middle line on the screen. |
| L | Move the cursor to the last line on the screen.   |

Table 70: Screen-oriented positioning

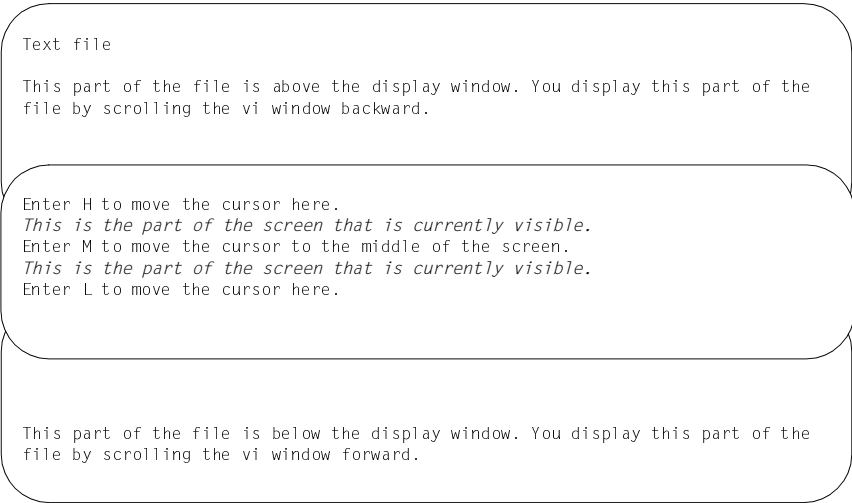


Figure 10: Screen-oriented positioning

Table 71 starting below summarizes the *vi* commands for positioning the cursor by character, line, word, sentence and paragraph or on the screen. Additional *vi* commands for positioning the cursor are described later on in this chapter

| Character-oriented positioning                                                      |                                                                                                                                                                                                                                  |
|-------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| h                                                                                   | Move the cursor one character to the left.                                                                                                                                                                                       |
| l                                                                                   | Move the cursor one character to the right.                                                                                                                                                                                      |
|    | Move the cursor one character to the left.                                                                                                                                                                                       |
|    | Move the cursor one character to the right                                                                                                                                                                                       |
| fx                                                                                  | Move the cursor to the right to the specified character <i>x</i> .                                                                                                                                                               |
| Fx                                                                                  | Move the cursor to the left to the specified character <i>x</i> .                                                                                                                                                                |
| tx                                                                                  | Move the cursor to the right, to the character just before the specified character <i>x</i> .                                                                                                                                    |
| Tx                                                                                  | Move the cursor to the left, to the character just after the specified character <i>x</i> .                                                                                                                                      |
| ;                                                                                   | Continue searching in the same direction for the last character specified with <i>f</i> , <i>F</i> , <i>t</i> or <i>T</i> . The (;) command remembers the character and finds the next occurrence of it on the current line.     |
| ,                                                                                   | Continue searching in the opposite direction for the last character specified with <i>f</i> , <i>F</i> , <i>t</i> or <i>T</i> . The (,) command remembers the character and finds the next occurrence of it on the current line. |
| k                                                                                   | Move the cursor up to the same column in the previous line (if there is a character in that column).                                                                                                                             |
| j                                                                                   | Move the cursor down to the same column in the next line (if there is a character in that column).                                                                                                                               |
| -                                                                                   | Move the cursor to the start of the previous line.                                                                                                                                                                               |
|  | Move the cursor to the start of the next line.                                                                                                                                                                                   |

| Moving the cursor on the current line |                                                              |
|---------------------------------------|--------------------------------------------------------------|
| \$                                    | Move the cursor to the last character on the line.           |
| O                                     | Move the cursor to the first character on the line.          |
| ^                                     | Move the cursor to the first nonblank character on the line. |

| Word-oriented positioning |                                                                                                                                                                                                             |
|---------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| w                         | Move the cursor to the first character of the next word.                                                                                                                                                    |
| W                         | Move the cursor to the start of the next word. This command recognizes only white space as a delimiter and ignores all other punctuation.                                                                   |
| b                         | Move the cursor to the start of the previous word.                                                                                                                                                          |
| B                         | Move the cursor to the start of the previous word, interpreting only white space as a delimiter.                                                                                                            |
| e                         | Move the cursor to the end of the current word.                                                                                                                                                             |
| E                         | Move the cursor to the end of the current word. This command interprets only white space as a delimiter. It places the cursor on the last character before the next white space, or at the end of the line. |

| Sentence-oriented positioning |                                                       |
|-------------------------------|-------------------------------------------------------|
| (                             | Move the cursor to the start of the current sentence. |
| )                             | Move the cursor to the start of the next sentence.    |

| Paragraph-oriented positioning |                                                        |
|--------------------------------|--------------------------------------------------------|
| {                              | Move the cursor to the start of the current paragraph. |
| }                              | Move the cursor to the start of the next paragraph.    |

| Screen-oriented positioning |                                                   |
|-----------------------------|---------------------------------------------------|
| H                           | Move the cursor to the first line on the screen.  |
| M                           | Move the cursor to the middle line on the screen. |
| L                           | Move the cursor to the last line on the screen.   |

Table 71: Summary of cursor positioning commands

## 6.8 Positioning the cursor in undisplayed text

So far you have used commands that enable you to move the cursor within the window currently displayed on the screen.

*vi* also allows you to position the cursor at a point in the file that is not displayed in the current window. You are already familiar with two of these commands, *j* and *k*. If, however, you are editing a large file, this is a relatively slow and inaccurate method of positioning the cursor to a specific place. This section describes commands that help you move around a file more easily by allowing you to

- scroll forward or backward
- move the cursor to a specific line
- search for a pattern

## 6.8.1 Scrolling text

You are provided with four commands for scrolling the text in a file:

You scroll the screen forward using `CTRL f` and `CTRL d`, and backward using `CTRL b` and `CTRL u`.

### The `CTRL f` command

You use the `CTRL f` command to scroll the text forward by one full screen page. To do this, `vi` clears the screen display and redraws it completely. The two lines that were at the bottom of the original window are placed at the top of the new window. If too few lines are left in the file to fill the window, a tilde (~) is displayed at the start of each empty line.

`vi` clears and redraws the screen as follows:

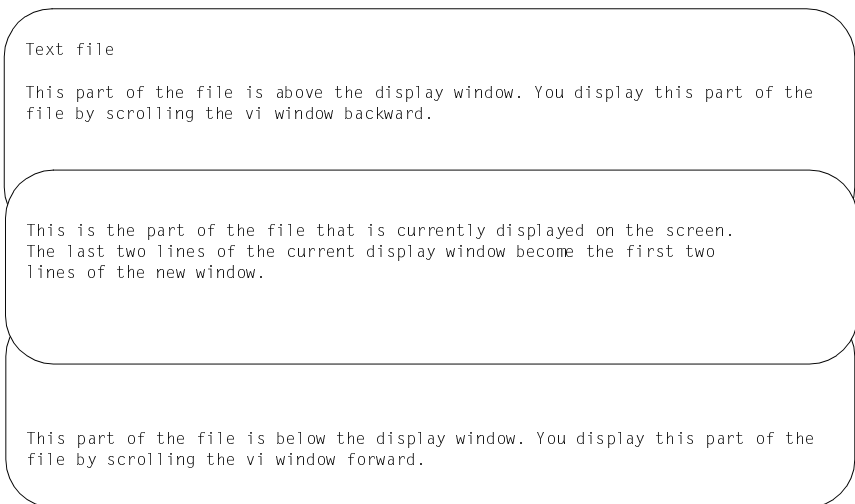


Figure 11: Scrolling forward

### The `CTRL d` command

You use the `CTRL d` command to scroll the screen display forward by half a screen page. If, when you enter `CTRL d`, the cursor is located on the last line of the file, an audible signal is issued and the current screen display remains unchanged.

### The **CTRL** b command

You use the **CTRL** b command to scroll the screen display backward by a full screen page. In other words, you display the previous screen page. To do this, vi clears the screen and redraws it. The text that was originally above the current screen page is now displayed on the screen.

Unlike the **CTRL** f command, the **CTRL** b command does not display any of the lines from the original screen display. If not enough lines are above the current window to fill a full new window, an audible signal is issued and the current screen display remains unchanged.

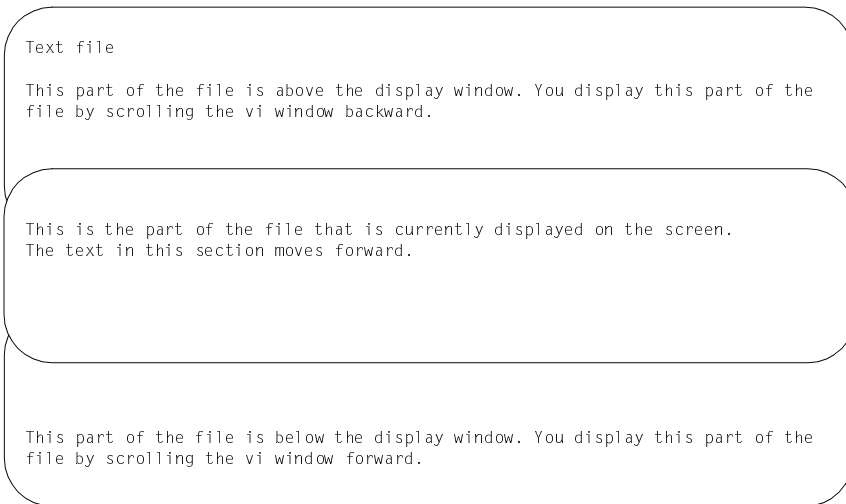


Figure 12: Scrolling backward

### The **CTRL** u command

You use the **CTRL** u command to scroll the screen display backward by half a screen page. The lines that were originally above the screen display now appear on the screen. The lines at the bottom of the screen are moved forward.

Now scroll the file backward so that the text above the current screen display becomes visible:

**CTRL** u

As soon as the cursor reaches the start of the file, an audible signal is issued to indicate that you cannot scroll any further in this direction.



### 6.8.2 Moving the cursor to a specific line

You use the G command to move the cursor to a specific line in the text. If this line is not in the current window, G clears and redraws the screen. The line you specified is now displayed in the middle of the screen. If you do not specify a line, the G command moves the cursor to the last line in the file

|    |                                                      |
|----|------------------------------------------------------|
| G  | Move the cursor to the last line in the file         |
| nG | Move the cursor to the <i>n</i> th line in the file. |

Table 72: Moving the cursor to a specific line

### 6.8.3 Line numbers

The lines of a file are numbered according to their position in the buffer. You can find out the number of a particular line by moving the cursor to this line and entering

`[CTRL] g`. This command issues a status message at the bottom of the screen stating

- the name of the file,
- whether the buffer has been modified since it was last written to a file,
- the number of the line on which the cursor is currently positioned,
- the total number of lines in the buffer, and
- the position of the current line relative to the total contents of the buffer, specified in percent.

## 6.8.4 Searching for patterns (the / and ? commands)

So far you have learned about methods for moving the cursor effectively around the text. If you are looking for a specific place in your text, however, the fastest way to find it is to use one of the following commands: /, ?, n or N.

These commands allow you to search forward or backward in the buffer for the next occurrence of a specified character pattern. The / and ? commands, unlike other *vi* commands, appear on the status line as you type them in. You can use the n and N commands to repeat the last search operation.

You can specify / followed by a pattern (*/pattern*) to search forward in the buffer for the next occurrence of the characters in the pattern. This command moves the cursor to the first of these characters. For example, the command line

/go now 

searches the buffer for the next occurrence of the string *go now*. If the search is successful, the cursor moves to the first letter of this string.

If the search pattern consists of two or more words, these words must all be on the same line. If, for example, the word *go* is at the end of one line and the word *now* is at the start of the next when you are searching for *go now*, the search command will not find that occurrence.

However, this search command does wrap around the end of the buffer to continue a search. If, for example, you reach the end of the buffer and the pattern you are searching for (using the */pattern* command) is located at the start of the buffer, the command will find the pattern.

? followed by a pattern (*?pattern*) searches the buffer backward for the first occurrence of the characters in the pattern. The cursor then moves to the first of these characters. For example, the command line

?stood up 

searches the buffer for the first occurrence of the string *stood up* above the current cursor position. If the search is successful, the cursor moves to the *s* of *stood*.

The *?pattern* command also requires that, if the search pattern contains two or more words, they must all be located on the same line in the text. This search also wraps around the beginning of the buffer.

The `n` and `N` commands allow you to continue searches you have requested with `/pattern` or `?pattern` without reentering them.

|                |                                                           |
|----------------|-----------------------------------------------------------|
| <code>n</code> | Repeat the last search command.                           |
| <code>N</code> | Repeat the last search command in the opposite direction. |

Table 73: Command repetition

For example, you want to search forward for the seven-letter pattern *mustard*. You initiate the search using `/mustard` and then continue it using `n`. The following example gives a step-by-step illustration of a forward search operation using the `n` command that finds three occurrences of the character string *mustard*:

There are more than 250 different types of mustard, 150 of which are made in France alone. The British, too, have invented mustards of a quality comparable to that of French mustard.

The basic ingredient of any mustard is the mustard seed. Mustard seeds may be left whole or ground into a fine or coarse powder.

~

~

~

~

`/mustard` 

There are more than 250 different types of mustard, 150 of which are made in France alone. The British, too, have invented mustards of a quality comparable to that of French mustard.

The basic ingredient of any mustard is the mustard seed. Mustard seeds may be left whole or ground into a fine or coarse powder.

~

~

~

~

n

There are more than 250 different types of mustard, 150 of which are made in France alone. The British, too, have invented mustards of a quality comparable to that of French mustard.

The basic ingredient of any mustard is the mustard seed. Mustard seeds may be left whole or ground into a fine or coarse powder.

~  
~  
~

n

There are more than 250 different types of mustard, 150 of which are made in France alone. The British, too, have invented mustards of a quality comparable to that of French mustard.

The basic ingredient of any mustard is the mustard seed. Mustard seeds may be left whole or ground into a fine or coarse powder.

~  
~  
~

The / and ? search commands do not allow you to specify particular occurrences of a pattern. You cannot, for example, request the third occurrence of a pattern (from the current cursor position).

Table 74 summarizes the vi commands available for positioning the cursor (by scrolling the text, specifying a line number and searching for a pattern).

| Scrolling                      |                                                                                                                                |
|--------------------------------|--------------------------------------------------------------------------------------------------------------------------------|
| <code>[CTRL] f</code>          | Scroll the screen forward by a full screen page, displaying the text below the current screen display.                         |
| <code>[CTRL] d</code>          | Scroll the screen forward by half a screen page, displaying the text below the current screen display.                         |
| <code>[CTRL] b</code>          | Scroll the screen backward by a full screen page, displaying the previous screen page.                                         |
| <code>[CTRL] u</code>          | Scroll the screen backward by half a screen page, displaying the previous screen page.                                         |
| Positioning on a numbered line |                                                                                                                                |
| <code>1G</code>                | Move the cursor to the first line of the file.                                                                                 |
| <code>nG</code>                | Move the cursor to the nth line of the file                                                                                    |
| <code>G</code>                 | Move the cursor to the last line of the file.                                                                                  |
| <code>[CTRL] g</code>          | Output the line number and other information on the file.                                                                      |
| Searching for patterns         |                                                                                                                                |
| <code>/muster</code>           | Search forward in the buffer for the next occurrence of the pattern. The cursor moves to the first character of the pattern.   |
| <code>?muster</code>           | Search backward in the buffer for the first occurrence of the pattern. The cursor moves to the first character of the pattern. |
| <code>n</code>                 | Repeat the last search command.                                                                                                |
| <code>N</code>                 | Repeat the last search command in the opposite direction.                                                                      |

Table 74: Summary of additional cursor positioning commands

Now try Exercise 3 at the end of this chapter.

## 6.9 Deleting text

You can use various commands to delete text in command mode. In input mode, however, you can only undo the entry of small amounts of text. Also note that you can undo all the effects of your most recent command.

### 6.9.1 Deleting text in input mode

When working in input mode, you can delete a character at any time using .

|                                                                                   |                                                                                          |
|-----------------------------------------------------------------------------------|------------------------------------------------------------------------------------------|
|  | Delete the character you have just entered (to the left of the current cursor position). |
|-----------------------------------------------------------------------------------|------------------------------------------------------------------------------------------|

Table 75: Deleting text using BACKSPACE

When you press  in input mode, you move the cursor to the left, one character at a time. You delete each character that the cursor backs across.

**Note:**

Do not get annoyed by the fact that the characters you delete are still visible on the screen. These characters do not disappear from the screen until you type over them or press  to return to command mode.

Ingredients:

One large cucumber, peeled, seeded and diced  
A small onion, chopped  
Three tomatoes, peeled and diced  
One small pepper, seeded and diced  
Four tablespoons olive oil  
Garlic to taste  
Lemon juiceeg

By pressing **[X]** twice, you delete the last two characters to the left of the cursor:

```
Ingredients:
One large cucumber, peeled, seeded and diced
A small onion, chopped
Three tomatoes, peeled and diced
One small pepper, seeded and diced
Four tablespoons olive oil
Garlic to taste
Lemon juiceee
```

When you press **[ESC]**, the two characters you have deleted disappear from the screen and the cursor moves to the last character of the line:

```
Ingredients:
One large cucumber, peeled, seeded and diced
A small onion, chopped
Three tomatoes, peeled and diced
One small pepper, seeded and diced
Four tablespoons olive oil
Garlic to taste
Lemon juice
```

There is another way of deleting text in input mode. The key combination **[CTRL] w** moves the cursor back to the first character of the last word you entered. You then delete this word from the screen by pressing **[ESC]** or overwriting the original characters with new text.

|                 |                                             |
|-----------------|---------------------------------------------|
| <b>[CTRL] w</b> | Back up over the word you have just entered |
|-----------------|---------------------------------------------|

Table 76: Deleting text using **[CTRL] w**

### 6.9.2 Undoing the last command

If you want to delete text, try using the `u` command. This command undoes the last command you issued

|                |                                                                         |
|----------------|-------------------------------------------------------------------------|
| <code>u</code> | Undo the last command.                                                  |
| <code>U</code> | Restores the current line to the state it was in before you changed it. |

Table 77: Undoing a command

If you delete one or more lines by accident and then enter the `u` command, your lines will reappear on the screen. You can also use `u` to undo the effects of any commands that you have entered by mistake and that have changed the contents of the editor buffer. However, you cannot use the `u` and `U` commands to undo positioning or search commands.

If you enter `u` twice in a row, the second `u` command will undo the first. In other words, you undo the `u` command itself.

If, for example, you delete a line accidentally and restore it using `u`, entering `u` a second time will delete the line again.

Provided that you have not moved the cursor to a different line in the meantime, you can use the `U` command to undo all the changes you have made to the current line.

### 6.9.3 Deleting text in command mode

As you already know, you can precede a command with a number. Many of the `vi` commands (such as those for deleting or overwriting text, for instance) also allow you to enter a cursor positioning command after another command. The positioning command can specify a text object, such as a word, line, sentence or paragraph. This type of `vi` command has the following general format:

```
[number]command[text_object]
```

The optional parts of this command format are enclosed in square brackets.

All the delete commands that you issue in command mode remove unwanted text from the screen. The delete command follows the general format of a `vi` command:

```
[number]d[text_object]
```



## Deleting words

You can delete a word or part of a word using the **d** (**delete**) command. Position the cursor on the first character you want to delete and enter **dw**. This deletes the character on which the cursor is positioned as well as all the remaining characters in the word.

Use **b** to move the cursor to the start of the word involved and delete it by entering:

**dw**

Ingredients:

One large cucumber, peeled, seeded and diced  
A small onion, chopped  
Three tomatoes, peeled and diced  
One small pepper, seeded and diced  
Four tablespoons olive oil  
Garlic to taste  
Lemon juice

Ingredients:

One large cucumber, peeled, seeded and diced  
A small onion, chopped  
Three tomatoes, peeled and diced  
One small pepper, seeded and diced  
Four tablespoons olive oil  
Garlic to taste  
Lemon juice

Ingredients:

One large cucumber, peeled, seeded and diced  
A small onion, chopped  
Three tomatoes, peeled and diced  
One small pepper, seeded and diced  
Four tablespoons olive oil  
Garlic to taste  
Lemon\_

You can use the `dw` command to delete a word or punctuation mark together with the blanks that follow it. If you want to delete more than one word or punctuation mark at once, prefix the command with a number. If, for example, you want to delete three words and two commas, enter the command `5dw`.

Deleting sentences

You delete sentences using the following commands:

|    |                              |
|----|------------------------------|
| d( | Delete the current sentence. |
| d) | Delete the next sentence.    |

Table 78: Deleting a sentence

If you want to use this command to delete entire sentences, you must make sure that the period at the end of every sentence is followed by two blanks. Otherwise `vi` is unable to recognize the end of a sentence.

Deleting paragraphs

You delete paragraphs using the following commands

|    |                               |
|----|-------------------------------|
| d{ | Delete the current paragraph. |
| d} | Delete the next paragraph.    |

Table 79: Deleting a paragraph

**Note:**

Remember that you can restore text immediately after deleting it by entering the `u` command, provided that you did not issue any other edit commands in the meantime.

**Deleting lines**

You delete a line by entering the `dd` command. You can delete several lines by prefixing the command with a number. For example, the command

```
10dd
```

deletes ten lines. Once you have deleted ten lines, `vi` issues the following message on the status line:

```
10 lines deleted
```

If the file contains fewer lines than you have specified in the delete command, an audible signal is issued and no lines are deleted.

**Deleting text from the cursor to the end of the line**

If you want to delete all the text between the current cursor position and the end of the line, position the cursor on the first character you want to delete and enter

`D` or `d$`.

These commands can only be applied to the current line. You cannot prefix them with a number specification.

Table 80 summarizes the *vi* commands for deleting text.

| Command                                                                             | Function                                                                                    |
|-------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------|
| In input mode:                                                                      |                                                                                             |
|    | Delete the character on which the cursor is currently positioned.                           |
|  w | Delete the current word.                                                                    |
| @                                                                                   | Delete the current line of text or all the new text you have entered on the current line.   |
| In command mode:                                                                    |                                                                                             |
| u                                                                                   | Undo the last edit command.                                                                 |
| U                                                                                   | Restore the original contents of the current line.                                          |
| x                                                                                   | Delete the current character.                                                               |
| ndx                                                                                 | Delete <i>n</i> text objects of type <i>x</i> .                                             |
| dw                                                                                  | Delete the text between the current cursor position and the next blank or punctuation mark. |
| dW                                                                                  | Delete the word or punctuation mark between the current cursor position and the next blank. |
| dd                                                                                  | Delete the current line.                                                                    |
| D                                                                                   | Delete the text between the current cursor position and the end of the line.                |
| d)                                                                                  | Delete the text between the current cursor position and the end of the current sentence.    |
| d}                                                                                  | Delete the text between the current cursor position and the end of the current paragraph.   |

Table 80: Summary of commands for deleting text

Now try Exercise 4 at the end of this chapter.

## 6.10 Modifying text

So far you have used several methods to delete sections of text and add to existing text. You now know how to modify text using the delete commands and the commands available in input mode.

*vi* also provides you with three basic commands that allow you to overwrite characters, i.e. to delete and create text at the same time:

- **r** (replace)
- **s** (substitute)
- **c** (change)

### 6.10.1 Replacing text (the **r** command)

The **r** command replaces the current character with the next character you enter

|                         |                                                                                                                                                                                                                          |
|-------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>rx</b>               | Replace the character on which the cursor is positioned with <i>x</i> . Because this command does not initiate input mode, you do not need to press <b>ESC</b> afterwards.                                               |
| <b>nr<math>x</math></b> | Replace <i>n</i> characters to the right of the current cursor position with <i>x</i> . This command terminates automatically after replacing the <i>n</i> th character. You do not need to press <b>ESC</b> afterwards. |
| <b>R</b>                | Overwrite any number of characters to the right of the current cursor position. If the new characters do not all fit on the current line, the command appends these characters as new text.                              |

Table 81: Replacing text using **r**

Let us assume that you want to change the word *acts* to *ants* in the following sentence:

The circus has many acts.

Position the cursor on the letter *c* in the word *acts* and enter

**rn**

The modified sentence looks like this:

The circus has many ants.

If you want to change *ants* to *XXXX*, move the cursor to the letter *a* in the word *ants* and enter

4rX

The *r* command replaces each of the letters in the word *ants* with the letter *X*.

The circus has many XXXX.

### 6.10.2 Substituting text (s command)

The *s* command, too, enables you to replace text. However, it differs from the *r* command in that it allows you to continue to insert text from that point until you press `[ESC]`.

|    |                                                                                                                                  |
|----|----------------------------------------------------------------------------------------------------------------------------------|
| s  | Delete the character on which the cursor is positioned and append text. You can exit input mode by pressing <code>[ESC]</code> . |
| nS | Delete <i>n</i> characters and append text. You can exit input mode by pressing <code>[ESC]</code> .                             |
| S  | Substitute all characters on the line.                                                                                           |

Table 82: Substituting text using *s*

When you enter the *s* command, the last character in the string to be replaced is overwritten by a \$ sign. The characters to be replaced are not deleted from the screen until you overwrite them or exit input mode by pressing `[ESC]`. Note that you cannot specify a text object as an argument for *r* or *s*.

Let us assume that you want to substitute the word *hundred* for the word *million* in the sentence *I weigh one million kilos*. Position the cursor on the letter *m* in the word *million* and enter *7s*. Notice where the \$ appears:

I weigh one million kilos\_

b

I weigh one million kilos

7s

I weigh one millio\$ kilos

I weigh one hundred\_kilos

### 6.10.3 Changing text (c command)

You can use the `c` command to overwrite individual characters, words or entire lines and then continue entering text. Once you have entered this command, press `[ESC]`.

|                  |                                                                                                                                                                   |
|------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>nCx</code> | Overwrite <i>n</i> text objects of type <i>x</i> , e.g. sentences, specified in parentheses ( <i>()</i> ), and paragraphs, specified in braces ( <i>{}</i> ).     |
| <code>cw</code>  | Overwrite a word or the remaining characters of a word with new text. The <i>vi</i> editor uses <code>\$</code> to indicate the last character to be overwritten. |
| <code>nCW</code> | Overwrite <i>n</i> words.                                                                                                                                         |
| <code>cc</code>  | Overwrite all the characters on the line.                                                                                                                         |
| <code>nCC</code> | Overwrite all the characters on the line and up to <i>n</i> lines of text.                                                                                        |
| <code>C</code>   | Overwrite the characters between the current cursor position and the end of the line.                                                                             |
| <code>nC</code>  | Overwrite the characters between the current cursor position and the end of the line, as well as all lines following the current line (up to <i>n</i> lines).     |

Table 83: Changing text

When you issue the `c` command, the last character to be overwritten is replaced with `$`. You position the cursor at the start of the word you want to overwrite.

```
Little Miss Muffet sat on a tuffet.
Eating her curds and whey;
Along came a spider and sat down beside her
And frightened Miss Muffet away.
```

To change the word *sat* to *reclined*, enter

`cw`  
and then overwrite *sat* with *reclined*.



```
Little Miss Muffet sa$ on a tuffet.
Eating her curds and whey;
Along came a spider and sat down beside her
And frightened Miss Muffet away.
```

```
Little Miss Muffet reclined_on a tuffet.
Eating her curds and whey;
Along came a spider and sat down beside her
And frightened Miss Muffet away.
```

Note that the new word (*reclined*) contains more letters than the word you want to overwrite (*sat*). Once you have executed the command, return *vi* to input mode by pressing **[ESC]**.

The **C** command works in the same way except that it overwrites the rest of the current line. When you enter the command, it uses **\$** to mark the end of the text to be overwritten. *vi* returns to input mode when you press **[ESC]**. You can now enter new text over the old.

You can also specify a combination of these arguments. For example, type in the following command and see what happens:

```
c{
```

Experiment with other combinations of arguments. Try to prefix the commands with a number. If you make a mistake, simply undo your changes using the **u** command. Before you enter the **u** command, press **[ESC]**, since the **c** command puts you in input mode.

Compare the functions of the **S** and **cc** commands. Both produce the same results.

Table 84 summarizes the *vi* commands for overwriting text.

| Command    | Function                                                                                                                                 |
|------------|------------------------------------------------------------------------------------------------------------------------------------------|
| r          | Replace the current character.                                                                                                           |
| R          | Replace only those characters that you entered before pressing [ESC].                                                                    |
| s          | Delete the character on which the cursor is currently positioned and append text. You exit input mode by pressing [ESC].                 |
| S          | Substitute all characters on a line.                                                                                                     |
| cc         | Overwrite all the characters on the line                                                                                                 |
| <i>ncx</i> | Overwrite <i>n</i> text objects of type <i>x</i> , e.g. sentences (specified using parentheses) and paragraphs (specified using braces). |
| cw         | Overwrite one word or the remaining characters of a word.                                                                                |
| C          | Overwrite the characters between the current cursor position and the end of the line.                                                    |

Table 84: Summary of commands for overwriting text

## 6.11 Cutting and pasting text

*vi* provides a set of commands that enable you to cut sections of text from a file and paste them in. Other commands allow you to copy a section of text and paste it to a different part of the file.

You perform these text operations by placing the text in a temporary buffer. The sections of text that you copy are stored here until you are ready to process them further. You already used this kind of temporary buffer in conjunction with the *u* and *U* commands. It would not be possible to restore deleted text passages without the help of buffers.

6.11.1 Moving text

This section describes the procedure for moving sections of text to a different location. First delete the lines you want to move, using the `dd` command, for instance. The text you last deleted is automatically stored in a temporary buffer. You then move the cursor to the place where you want to insert the deleted lines and press `p` (for **put**). The deleted lines then appear below the current line

|   |                                                                                                                   |
|---|-------------------------------------------------------------------------------------------------------------------|
| p | Paste the contents of the temporary buffer to the right of the current cursor position or below the current line. |
|---|-------------------------------------------------------------------------------------------------------------------|

Table 85: The temporary buffer p

Part of a line deleted using the `D` command can be pasted into any other line. To do this, position the cursor on a blank between two words and press `p`. The partial line is pasted to the right of the current cursor position.

Characters that you delete by specifying the `x` command with a number of characters are also stored in a temporary buffer. You can use the `p` command to paste any text object that you just deleted to any other location in the file.

You must use the `p` command immediately after deleting a section of text, since the temporary buffer can only store the result of one command. The `p` command also allows you to paste the copy of a text block, placed in the temporary buffer using the `y` command (**y**ank), to any other location in the file (the `y` command is described in more detail in section 6.11.3 *Copying text*).

6.11.2 Correcting transposed characters

When entering text in a hurry, you are sure to transpose some of the letters. A quick way to correct transposed letters is to combine the `x` and `p` commands as `xp`. The `x` command deletes the letter and `p` pastes it after the next character.

Notice the error in the following line:

```
This is not a reciepe_
```

You can correct this error quickly by positioning the cursor on the letter e in *recie*p and then pressing x and p one after the other.

```
This is not a recie
```

The line of text then looks like this:

```
This is not a recip
```

6.11.3 Copying text

vi enables you to save (copy) one or more lines of text to a temporary buffer using the y command (yank). You can then paste a copy of this text block to any location in the file. To paste the copy, move the cursor to the desired location and enter the p command. This command pastes the text onto the following line.

The y command follows the general format of a vi command:

```
[number]y[text_object]
```

The y command does not delete the lines from their original position in the file. Instead, it provides you with a convenient way to paste a particular text block more than once in a file without having to enter it again each time.

Table 86 summarizes the ways in which you can use the y command.

| Command | Function                                                                    |
|---------|-----------------------------------------------------------------------------|
| nyx     | Copy n text objects of type x to a temporary buffer.                        |
| yw      | Copy a word to the temporary buffer.                                        |
| yy      | Copy the current line to the temporary buffer.                              |
| nyy     | Copy n lines to the temporary buffer.                                       |
| y)      | Copy all characters up to the end of the sentence to the temporary buffer.  |
| y}      | Copy all characters up to the end of the paragraph to the temporary buffer. |

Table 86: Summary of temporary buffers and the y command

## 6.11.4 Copying or moving text using named buffers

Moving or copying several sections of text is tedious work if you use only the methods described so far in this section. *vi* does, however, provide you with a quick and easy way of copying sections of text: named buffers. You can use a named buffer to store text until you want to paste it into the file. You can either copy the desired text to the buffer (using the *y* command) or delete it from its original position.

Named buffers are particularly useful if you want a text block to appear in several places in the file. The extracted text is held in the specified buffer until you terminate the editing session or store another section of text in this buffer (using the *y* or *d* command).

The command has the following general format:

`[number][x]command[text_object]`

*x* is the name of the buffer, and can consist of any number of lowercase letters. The buffer name must be prefixed with a double quote (") so that it can be recognized as a buffer name.

For example, position the cursor at the start of a line and enter

```
3 " ayy
```

Enter more text and move the cursor to the end of the line. Enter

```
" ap
```

then check whether the lines that you saved in the buffer *a* have been pasted to the end of the file.

Table 87 summarizes the commands for cutting and pasting text.

| Command | Function                                                                                                                      |
|---------|-------------------------------------------------------------------------------------------------------------------------------|
| p       | Paste the contents of the temporary buffer (the text you last deleted or copied) to the right of the current cursor position. |
| yy      | Copy a line of text to a temporary buffer.                                                                                    |
| nyx     | Copy <i>n</i> text objects of type <i>x</i> to a temporary buffer.                                                            |
| "xyn    | Copy a text object of type <i>n</i> to the named buffer <i>x</i> .                                                            |
| "xp     | Paste the contents of buffer <i>x</i> to the right of the current cursor position.                                            |

Table 87: Summary of commands for cutting and pasting text

Now try Exercise 5 at the end of this chapter.

## 6.12 Special commands

You will find the following special commands useful when working with *vi*:

|                 |                                                       |
|-----------------|-------------------------------------------------------|
| .               | Repeat the last command.                              |
| nJ              | Join <i>n</i> lines together.                         |
| <div>CTRL</div> | Clear and redraw the screen.                          |
| ~               | Change lowercase letters to uppercase and vice versa. |

Table 88: Summary of special commands

### 6.12.1 Repeating the last command

You can specify the period (.) to repeat the last command you used to add, delete or modify text.

Let us assume, for example, that you forgot to capitalize the *S* in *United States*. However, you do not want to capitalize the *s* in *chemical states*. One way to correct this problem is by searching for the word *states*. The first time you find it in the expression *United States*, you can correct it. Then continue your search.

When you find another occurrence, you can simply enter a period (.). *vi* will remember your last command and repeat the substitution of the uppercase *S* for the lowercase *s*.

### 6.12.2 Joining lines together

You can join lines together using the `J` command. Place the cursor anywhere on the first of the two lines you want to join together and enter the `J` command. The current line is joined with the line that follows it.

Let us assume, for example, that your file contains the following two lines of text:

```
Dear Mr.
Smith:
```

To join these two lines together, place the cursor anywhere on the first line and enter

```
J
```

The following line is displayed on your screen immediately:

```
Dear Mr. Smith:
```

*vi* automatically inserts a blank between the last word of the first line and the first word of the second line.

### 6.12.3 Clearing and redrawing the screen

If another Reliant UNIX user sends you a message using the *write* command during a *vi* session, the message appears in your current window, obscuring part of the text you are currently editing on the screen, although the contents of the text buffer remain intact.

To restore the current screen display, you must switch *vi* to command mode before entering `[CTRL] L`. *vi* clears the message from the screen and redraws the display exactly as it was before the message arrived.

## 6.12.4 Changing lowercase to uppercase and vice versa

The quickest way to change any uppercase letter to the corresponding lowercase letter and vice versa is to place the cursor on it and enter a tilde (~).

For example, to change the lowercase letter *a* to the uppercase *A*, enter the tilde. You can change several letters in this way, by either entering the tilde several times or prefixing the command with a number.

## 6.13 Line editor commands

When you use *vi*, you also have access to a set of commands provided by the *ex* line editor. All the *ex* commands are listed under *ex* in the *Commands* manual [5]. The following sections describe some of the most commonly used line editor commands.

Line editor commands begin with a colon (:). When you enter a colon, the cursor moves to the status line and displays a colon prompt. The command you enter is also displayed.

### 6.13.1 Returning temporarily to the shell

Once you have called *vi*, the buffer contents fill the entire screen and you cannot execute any shell commands. In some situations, however, you may want to do so. For example, you may want to fetch information from another file and incorporate it into your current text. You could do this by calling a shell command that displays the contents of a text file (e.g. *cat* or *pg*). This is a time consuming and tedious process, however, since it involves quitting and restarting the editor.

*vi* provides you with two options for exiting the editor temporarily so that you can call shell commands (and even edit other files) without having to write your buffer contents and quit the editor. These commands are `:!` and `:sh`.

The `:!` command allows you to exit the editor temporarily and issue a shell command. To do this, you enter the character combination `:!` in *vi* command mode. These characters appear at the bottom of the screen. You enter a shell command immediately after the `!` command. The shell calls the specified command, outputs any results and issues the message:

```
[Hit return to continue]
```



As soon as you press `␣`, vi redraws the screen and the cursor reappears in the same position as before you exited the editor.

The *ex* command `:sh` has a similar function. It enables you to exit vi temporarily and opens a new shell. You enter `:sh` in vi command mode and press `␣`. The shell prompt appears on the next line. You can enter your commands after this prompt, just as if you were working at the level of the first shell. When you want to return to your vi session, enter `CTRL d` or *exit* to exit the shell. The screen is redrawn (with the contents of the buffer) and the cursor reappears in its original position.

Even if you enter a command to move to a different directory while you are temporarily in the shell, the contents of the vi buffer that you were processing before you exited the editor are redisplayed on the screen when you enter *exit* or `CTRL d`.

### 6.13.2 Writing text to a file

If you called vi with a file name, you can write the contents of the current editor buffer to this file at any time using the `:w` command (write). Simply enter

```
:w ␣
```

You can also write the buffer contents or a section of them to a different file, or create a new file for this purpose. To create a new file, you specify a line or a range of lines (with their line numbers), together with the name of the new file. You can specify as many lines as you like. The command has the following general format:

```
:[line_number][,line_number]w [file_name]
```

If, for example, you want to write the third line of the buffer to a file called *three*, enter

```
:3w three ␣
```

vi informs you that you have created a new file by issuing the following message:

```
„three“ [New file] 1 line, 20 characters
```

To write your current line to a file, you can specify a period as the line address:

```
:.w temp ␣
```

A new file called *temp* is created in your current directory. This file contains only the current line in the *vi* buffer.

If you want to write a whole section of the buffer to a new file, you can specify a range of lines. For example, to write lines 23 through 37 to a file, enter

```
:23,37w new_file ↵
```

### 6.13.3 Positioning to a specific line

You can position the cursor to any line by entering the `:` command together with a line number. The command line

```
:n ↵
```

positions the cursor to the *n*th line in the buffer.

### 6.13.4 Deleting the remaining lines in the buffer

One of the easiest ways of deleting all the lines between the current line and the end of the buffer is to issue the line editor command `d`, specifying the metacharacters that represent the current line and the last line:

```
:$d ↵
```

The period stands for the current line while `$` stands for the last line.

### 6.13.5 Reading a file into the buffer

If you want to read in text from another file and paste it after a specific line in the editing buffer, you use the `:r` command (*read*).

Let us assume, for example, that you want to incorporate the contents of a file called *data* into your current file. First position the cursor on the line above the place where you want to paste the contents of *data*. Then enter

```
:r data ↵
```

Alternatively, you can specify the line number. If, for example, you want to paste the *data* file below line 56 in the buffer, enter

```
:56r data ↵
```

**Note:**

Remember that you can use the `u` command to undo *ex* commands.

6.13.6 Global changes

One of the most powerful *ex* commands is `g` (**g**lobal). You can use this command to search for a specific pattern in a text and process this text using another command.

You must first familiarize yourself with the following set of commands

|                                      |                                                                                                                                                                                                                                                                      |
|--------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>:g/pattern/command ↵</code>    | The <i>ex</i> command <i>command</i> is executed for each line that contains the <i>pattern</i> .                                                                                                                                                                    |
| <code>:s/pattern/new_text ↵</code>   | The line editor searches the file for the first occurrence of the <i>pattern</i> in the current line, and replaces these characters with the <i>new_text</i> .                                                                                                       |
| <code>:s/pattern/new_text/g ↵</code> | If you enter a lowercase <i>g</i> after the final delimiter in this command line, <i>ex</i> replaces every, occurrence of the <i>pattern</i> in the current line. Without the <i>g</i> command, <i>ex</i> replaces only the first occurrence of the <i>pattern</i> . |

Table 89: Replacing text

Let us assume that you have several pages of *Vegetarian recipes* and you want to change every occurrence of the word *lentil* to the words *red lentil*. You can do this in a single command line using the global command of the *ex* editor.

```
:g/lentil/s//red lentil/g ↵
```

This command line tells *ex* to search for the word *lentil* and replace each occurrence of it with the two words *red lentil*. You do not have to retype the word *lentil* between the two delimiters that follow the `s` command, since the command automatically uses the word you specified between delimiters after the global command `g`.

Table 90 summarizes the line editor commands available under vi.

| Command                    | Function                                                                                                                            |
|----------------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| :                          | The commands you enter after the colon are line editor commands.                                                                    |
| :sh ↵                      | Exit the editor temporarily in order to execute shell commands.                                                                     |
| ␣ d                        | Exit the shell in which you were working temporarily and return to the current vi window, where you can continue editing your file. |
| :n ↵                       | Position the cursor to the <i>n</i> th line of the buffer.                                                                          |
| :x,yw file ↵               | Write the lines in the range <i>x</i> through <i>y</i> to a new file.                                                               |
| :\$ ↵                      | Position the cursor to the last line of the buffer.                                                                                 |
| .,\$d ↵                    | Delete all lines between the current line and the end of the buffer.                                                                |
| :r shell.file ↵            | Paste the contents of the <i>shell.file</i> below the current line in the buffer.                                                   |
| :s/old_text/new_text/ ↵    | Replace the first occurrence of <i>old_text</i> in the current line with <i>new_text</i> .                                          |
| :s/old_text/new_text/g ↵   | Replace every occurrence of <i>old_text</i> in the current line with <i>new_text</i> .                                              |
| :g/old_text/s/new_text/g ↵ | Replace every occurrence of <i>old_text</i> in the file with <i>new_text</i> .                                                      |

Table 90: Summary of the line editor commands

## 6.14 Quitting vi

You are already familiar with some of the ways of quitting *vi*. You can also quit *vi* using the line editor commands that begin with a colon.

| Command                        | Function                                                                                                                                                                                                                                                                          |
|--------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| :wq ↵                          | Write the contents of the <i>vi</i> buffer to the Reliant UNIX file currently being edited, then quit <i>vi</i> .                                                                                                                                                                 |
| ZZ                             | Save the buffer contents only if you have made changes to them since issuing the last <i>w</i> command, then quit <i>vi</i> .                                                                                                                                                     |
| :w <i>file_name</i> ↵<br>:q ↵  | Write the contents of the temporary buffer to a new file called <i>file_name</i> , then quit <i>vi</i> .                                                                                                                                                                          |
| :w! <i>file_name</i> ↵<br>:q ↵ | Overwrite the existing file <i>file_name</i> with the contents of the buffer, then quit <i>vi</i> .                                                                                                                                                                               |
| :q! ↵                          | Quit <i>vi</i> without writing the buffer contents to a file. Any changes made to the buffer contents are lost.                                                                                                                                                                   |
| :q ↵                           | Quit <i>vi</i> without writing the buffer contents to a file. This command is only executed if you have not made any changes to the buffer contents. Otherwise <i>vi</i> tells you to save the buffer contents or quit the editor using :q! ↵ without saving the buffer contents. |

Table 91: Summary of line editor commands for quitting vi

The *:wq* and *ZZ* commands save the buffer contents to a file under the conditions described above, quit the *vi* editor and return you to the shell.

### Enter

:wq ↵

*vi* displays the name of the file and the number of lines and characters it contains.

How do you assign the file a different name? If, for example, you want to save the buffer contents to a new file called *test*, enter

:w test ↵

Once you have saved the buffer contents to the new file, you quit *vi* using the command:

```
:q 
```

If the file you specify already exists, the editor issues a warning. If, for example, you attempt to save the buffer contents to a file called *gazpacho*, the system issues the message:

```
"gazpacho" File exists - use "w! gazpacho" to overwrite
```

If you want to overwrite the contents of the existing file *gazpacho* with the buffer contents, use the `:w!` command:

```
:w! gazpacho 
```

The new file overwrites the existing file.

If you are editing a file called *memo* and you make changes that you do not want to save, you can quit *vi* without saving the buffer contents:

```
:q! 
```

## 6.15 Special options

If you call *vi* from the Reliant UNIX command line, you are provided with special options that allow you to

- restore a file after a system crash
- place several files in the editor buffer and edit them consecutively
- open a file and read it without the risk of accidental modification

### 6.15.1 Restoring a file after a system crash

In the event of a system crash, e.g. due to a power failure, *vi* terminates without writing the contents of the buffer to a file. Because *vi* makes a copy of the buffer contents, however, you can restore the file when you next log in to Reliant UNIX. To do this, you use the *vi* option `-r` in the following format:

```
$ vi -r file_name
```

Most of the changes that you made to the *file\_name* file before the system crash are stored in the *vi* buffer. You can now either edit the file or save it and exit the editor. *vi* automatically saves the file with the original file name, unless you specify a different one.

## 6.15.2 Editing more than one file

You can also edit two or more files during the same *vi* session. To do so, enter the *vi* command together with the names of the files you want to edit:

```
$ vi file1 file2
```

*vi* tells you how many files you are going to edit:

```
2 files to edit
```

Once you have edited the first file, write the changes you have made to the buffer contents into the file *file1*:

```
:w 
```

The system responds to this command by displaying a message at the bottom of the screen, giving you the name of the file and its size (the number of lines and characters). You can now enter the *:n* command (**n**ext) to go on to edit the next file:

```
:n 
```

The system responds by displaying a message at the bottom of the screen, giving you the name of the next file to be edited and its size (the number of characters/lines).

## 6.15.3 Viewing file contents

If you want to check the contents of a file, you can use the powerful search and scroll functions provided by *vi*. You may want to protect yourself against accidentally modifying the file. If you call the file with *view*, a read-only variant of *vi*, the file has temporary write protection.

Table 92 summarizes the special *vi* options.

| Option                                 | Function                                                                                                 |
|----------------------------------------|----------------------------------------------------------------------------------------------------------|
| <code>vi file1 file2 file3 ↵</code>    | Load three files ( <i>file1</i> , <i>file2</i> and <i>file3</i> ) into the <i>vi</i> buffer for editing. |
| <code>:w ↵</code><br><code>:n ↵</code> | Save the current file and continue the <i>vi</i> session with the next file.                             |
| <code>vi -r file1 ↵</code>             | Restore the changes made to <i>file1</i> .                                                               |
| <code>view file ↵</code>               | View the file in read-only mode to protect it against accidental modification.                           |

Table 92: Summary of special options

Now try Exercise 6 at the end of this chapter.



## 6.16 Exercises

### 6.16.1 Exercise 1

1-1. Create a file called *ex1*.

1-2. Enter the following five lines of text:

```
be, or not to be, that is the question,
Whether 'tis nobler in the mind to suffer
Or to take arms against a sea of troubles,
No more, and by a sleep to say we end
That flesh is heir to; 'tis a consummation
```

1-3. Enter the following text above the last line:

```
The heart-ache, and the thousand natural shocks
```

1-4. Insert the following line of text above the third line:

```
The slings and arrows of outrageous fortune,
```

1-5. Enter the following line of text below the current line:

```
And by opposing, end them. To die, to sleep-
```

1-6. Enter the following line of text below the last line:

```
Devoutly to be wished to die to sleep.
```

1-7. Position the cursor to the first line and insert the word *To* in front of the word *be*.

Now try these six commands out on your own examples.

1-8. Quit *vi*.

## 6.16.2 Exercise 2

- 2-1. Call *vi* and enter the following lines in a file called *ex2*.

```
Humpty Dumpty sat on on a wall,
Humpty Dumpty had a great fall.
All the King's horses and all the King's men
```

- 2-2. Position the cursor to the first line of the file. Note that the cursor will only move to the last character of the previous line if this line is shorter than the current line. Now position the cursor to the nineteenth column from the left.

- 2-3. Delete the nineteenth and twentieth characters from the left and the blank that follows them.

- 2-4. Position the cursor to the last character of the last line of text.

- 2-5. Insert the following new line of text:

```
Couldn't put Humpty together again.
```

- 2-6. Write the buffer contents to a file and quit *vi*.

- 2-7. Call *vi* again and append a blank line and the following additional line of text to the *ex2* file:

```
A nursery rhyme.
```

What is displayed at the bottom of the screen after you call *vi* to edit the *ex2* file?

### 6.16.3 Exercise 3

- 3-1. Create a file called *ex3*. Number the lines from 1 through 50. Your file should look something like this:

```
1
2
3
.
.
.
48
49
50
```

- 3-2. Try using each of the following scroll commands. How many lines does the screen display scroll backward in each case?

|      |   |
|------|---|
| CTRL | f |
| CTRL | b |
| CTRL | u |
| CTRL | d |

- 3-3. Position the cursor to the end of the file. Insert the following line of text:

```
123456789 123456789
```

What number does the `7h` command place the cursor on? What number does the `3l` command place the cursor on?

- 3-4. Enter the `$` and `O` commands. What happens?
- 3-5. Position the cursor to the first nonblank character on the line. Position the cursor to the first character of the next word. Position the cursor to the first character of the previous word. Position the cursor to the end of the word.
- 3-6. Position the cursor to the first line of the file. Enter the commands that position the cursor to the middle of the screen, on the first line of the screen and on the last line of the screen.
- 3-7. Search for the number 8. Then search for the next occurrence of the number 8.

### 6.16.4 Exercise 4

- 4-1. Create a file called *ex4* and enter the following four (incorrect) lines of text:

```
Mary had a little lamb
Its fleece was white as the
And everywhere that Mary went
was sure to go.
```

- 4-2. Position the cursor to line 2 and append the following text to this line:

```
driven snow
```

Delete the word *driven* in append mode.

Delete the word *the* in command mode.

Is there another way of deleting the word *the*?

- 4-3. Enter the following text at the start of line 4:

```
The lamb
```

Delete the line.

Is there a way of deleting the contents of the line without removing the line itself?

Delete all the lines using a single command.

- 4-4. Exit the display editor and delete the empty file from your directory.

### 6.16.5 Exercise 5

- 5-1. Call *vi* with the *ex2* file that you created in Exercise 2.

Position the cursor to line 7 and change its contents to END OF THE FILE.

- 5-2. Write the first seven lines of the file to buffer *z*. Paste the contents of buffer *z* after the last line of text.

- 5-3. Position the cursor to line 7 and change its contents to *This is the seventh line*.

- 5-4. Move the cursor to the last line of the file. Overwrite *FILE* with *EXERCISE*. Replace *THE* with *THIS*.

## 6.16.6 Exercise 6

- 6-1. Place the files *ex1* and *ex2* in the *vi* buffer for editing. Save *ex1* and call the next file in the buffer (*ex2*).

Write *ex2* to a file called *test*.

Quit *vi*.

- 6-2. Try entering the following command:

```
$ vi ex*
```

What is the effect of this command? Try to quit all the files as quickly as possible.

- 6-3. View the contents of *ex2* in read-only mode.

Scroll the screen display forward one full window.

Scroll the screen display forward by half a window.

Scroll the screen display backward by a one full window.

Scroll the screen display backward by half a window.

Quit the editor and return to the shell.

## 6.17 Solutions

There is often more than one way to perform a task in *vi*. The following solutions are simply suggestions. If you prefer a different solution, use it. Any method that works is correct.

### 6.17.1 Exercise 1

- 1-1. Enter

```
$ vi ex1
```

- 1-2. Enter

```
be, or not to be, that is the question, 
Whether 'tis nobler in the mind to suffer 
Or to take arms against a sea of troubles, 
No more, and by a sleep to say we end
That flesh is heir to; 'tis a consummation 
```

**1-3. Enter**

0

The heart-ache, and the thousand natural shocks **[ESC]**

**1-4. Enter**

3G

oThe slings and arrows of outrageous fortune, **[↓] [ESC]**

**The contents of your file should now look like this:**

be, or not to be, that is the question,  
Whether 'tis nobler in the mind to suffer  
The slings and arrows of outrageous fortune  
Or to take arms against a sea of troubles,  
No more, and by a sleep to say we end  
The heart-ache, and the thousand natural shocks  
That flesh is heir to; 'tis a consummation

- 1-5.** *Or to take arms against a sea of troubles* is the current line. To insert a new line of text below this line, use the `o` command.

- 1-6.** The current line is *And by opposing, end them. To die, to sleep-*. If you enter `G`, the cursor moves to the last line of text.

You can use the `A` command to append text to a line. If you then press **[↓]**, you create a new line. Now append the following line: *Devoutly to be wished to die to sleep!*

You press **[ESC]** to exit input mode.

**1-7. Enter**

1G

/be

i

To  **[ESC]**

- 1-8.** The `ZZ` command writes the buffer contents to the *ex1* file and returns you to the shell.

## 6.17.2 Exercise 2

- 2-1. Call vi with the *ex2* file:

```
$ vi ex2
```

Then enter the *a* command and enter the following text in your file:

```
Humpty Dumpy sat on on a wall,
Humpty Dumpy had a great fall.
All the King's horses and all the King's men
```

- 2-2. Use the *k* and *h* commands to move the cursor to the start of the first occurrence of the word *on*.
- 2-3. Enter the *x* command three times.
- 2-4. Use the *j* and *l* commands to move the cursor to the end of the text.
- 2-5. Enter the *a* command and enter the following text:

```


```

```
Couldn't put Humpty together again.
```

- 2-6. Enter the following command:

```
ZZ
```

vi displays the following message at the bottom of the screen:

```
"ex2" 4 lines, 115 characters
```

- 2-7. Enter the following command at shell level:

```
$ vi ex2
```

Move the cursor to the end of the text using the *j* and *l* commands.

Enter the *a* command and type in:

```


```

```
A nursery rhyme.
```

Notice the message that the system outputs when you call *ex2*:

```
"ex2" 4 lines, 115 characters
```

6.17.3 Exercise 3

3-1. Enter

```
$ vi ex3
a 1
2
3
.
.
.
48
49
50
```

3-2. Enter

```
CTRL f
CTRL b
CTRL u
CTRL d
```

Notice that the line numbers have changed.

3-3. Enter

```
G
$
a
123456789 123456789
7h
3l
```

The 7h command moves the cursor to the number 2 in the second set of numbers. The 3l command moves the cursor to the number 5 in the second set of numbers.

3-4. The \$ command moves the cursor to the end of the line. The 0 command moves the cursor to the first character on the line.

3-5. Enter

```
^
w
b
e
```



## 3-6. Enter

```
1G
M
L
H
```

## 3-7. Enter

```
/8 ↵
n
```

## 6.17.4 Exercise 4

## 4-1. Create the new file at shell level using

```
$ vi ex4
```

Enter

```
A
Mary had a little lamb ↵
Its fleece was white as the ↵
And everywhere that Mary went ↵
was sure to go. ↵
```

## 4-2. Enter

```
2G
A
driven snow CTRL w ESC
```

Enter Fd to move the cursor to the letter d in *driven*. Then enter

```
dw
```

Another way of deleting the word *the* is to use the 4x command.

## 4-3. The cursor is on the second line. Enter

```
2j
I
The lamb ESC
dd
```

If you want to delete the line and leave it blank, enter

```
u
```

to undo the effect of the dd command.

```
O
```

If you enter the number zero 0, the cursor moves to the start of the line.

```
D
d1G
```

4-4. To save your changes and quit vi, enter

```
ZZ
```

To delete the file, enter:

```
$ rm ex4
```

## 6.17.5 Exercise 5

5-1. Enter

```
$ vi ex2
7G
cc
END OF THE FILE [ESC]
```

5-2. Enter

```
1G
7"zyy
G
"zp
```

5-3. Enter

```
7G
cc
This is the seventh line. [ESC]
```

5-4. Enter

```
G
2w
cw
EXERCISE [ESC]
2b
cw
THIS [ESC]
```

## 6.17.6 Exercise 6

### 6-1. Enter

```
$ vi ex1 ex2
:w ↵
:n ↵
:w test ↵
ZZ
```

### 6-2. Enter

```
$ vi ex*
```

vi displays the following message:

```
4 files to edit
```

and calls all files with names that begin with *ex*. To quit *vi*, enter

```
:q!
```

### 6-3. Enter

```
$ view ex4
CTRL f
CTRL d
CTRL b
CTRL u
:q ↵
```

## 6.18 vi command summary

This section summarizes the most commonly used *vi* commands according to function groups.

| Loading vi         |                                                                |
|--------------------|----------------------------------------------------------------|
| vi ↵               | Call <i>vi</i> with an empty editor buffer.                    |
| vi <i>file</i> ↵   | Load <i>vi</i> together with the <i>file</i> you want to edit. |
| view <i>file</i> ↵ | Load <i>vi</i> together with the <i>file</i> you want to view. |

| Quitting vi      |                                                                                                           |
|------------------|-----------------------------------------------------------------------------------------------------------|
| ZZ               | Quit <i>vi</i> and write the buffer contents to the edited file. This command has the same effect as :wq. |
| :w ↵             | Write the buffer contents to the edited file.                                                             |
| :w <i>file</i> ↵ | Write the buffer contents to <i>file</i> .                                                                |
| :q ↵             | Quit <i>vi</i> , provided you have saved your changes.                                                    |
| :q! ↵            | Quit <i>vi</i> without saving your changes.                                                               |
| :wq              | Quit <i>vi</i> and write the buffer contents to the edited file. This command has the same effect as ZZ.  |

| Positioning the cursor                                                            |                                         |
|-----------------------------------------------------------------------------------|-----------------------------------------|
| h                                                                                 | One character to left.                  |
| j                                                                                 | One line down in same column.           |
| +                                                                                 | One line down and to start of line.     |
|  | One line down and to start of line.     |
| k                                                                                 | One line up in same column.             |
| -                                                                                 | One line up and to start of line.       |
|                                                                                   | One character to right.                 |
| H                                                                                 | Top of screen.                          |
| M                                                                                 | Middle of screen.                       |
| L                                                                                 | Bottom of screen.                       |
| O                                                                                 | Left edge of screen.                    |
|                                                                                   | Left edge of screen.                    |
| \$                                                                                | Last character in line.                 |
|                                                                                   | First non-whitespace character in line. |
| w                                                                                 | Start of next word.                     |
| e                                                                                 | End of next word.                       |
| b                                                                                 | Start of previous word.                 |
| W                                                                                 | Start of next word (extended word).     |
| E                                                                                 | End of next word (extended word).       |
| B                                                                                 | Start of previous word (extended word). |

| Positioning the window                                                                |                                   |
|---------------------------------------------------------------------------------------|-----------------------------------|
|  d | Half a screen forward.            |
|  u | Half a screen backward.           |
|  f | One screen forward.               |
|  b | One screen backward.              |
|  e | One line forward.                 |
|  y | One line backward.                |
| z  | Current line to top of screen.    |
| z.                                                                                    | Current line to middle of screen. |
| z-                                                                                    | Current line to bottom of screen. |
| 1G                                                                                    | Start of file.                    |
| nG                                                                                    | Line n to middle of screen.       |
| G                                                                                     | End of file.                      |

| Deleting and restoring text |                                                       |
|-----------------------------|-------------------------------------------------------|
| x                           | Delete character at cursor position.                  |
| X                           | Delete character to left of cursor.                   |
| dd                          | Delete current line.                                  |
| dG                          | Delete from current line to end of file.              |
| dH                          | Delete from start of screen to current line incl.     |
| D                           | Delete rest of line from cursor position.             |
| u                           | Restore deleted text from standard buffer.            |
| U                           | Restore current line.                                 |
| p                           | Write buffer contents after current cursor position.  |
| P                           | Write buffer contents before current cursor position. |

| Changing text |                                                  |
|---------------|--------------------------------------------------|
| r             | Replace current character with next one entered. |
| R             | Overwrite text starting at current character.    |
| o             | Insert blank line below current line.            |
| O             | Insert blank line above current line.            |
| a             | Append text after current character.             |
| A             | Append text to end of line.                      |
| i             | Insert text before current cursor position.      |
| I             | Insert text at start of line.                    |

| Copying        |                                                             |
|----------------|-------------------------------------------------------------|
| Y              | Yank current line.                                          |
| yy             | Yank current line.                                          |
| Yp             | Duplicate current line.                                     |
| yyə            | Duplicate current line.                                     |
| :10,13t 20     | Copy lines 10 through 13 and insert below line 20.          |
| :.t.           | Duplicate current line.                                     |
| .t.            | Duplicate current line.                                     |
| :r <i>file</i> | Copy contents of <i>file</i> and insert below current line. |

| Cutting and pasting |                                        |
|---------------------|----------------------------------------|
| 3dd                 | Delete 3 lines including current line. |
| p                   | Insert saved lines.                    |
| "a3dd               | Delete 3 lines and save to buffer a.   |
| "ap                 | Insert saved lines below current line. |

| Searching and replacing |                                                                          |
|-------------------------|--------------------------------------------------------------------------|
| /xyz                    | Search forward through entire editor buffer for <i>xyz</i> .             |
| /xyz_ _/                | Search forward through entire editor buffer for <i>xyz</i> and 2 blanks. |
| :s/xyz/abc/             | Replace <i>xyz</i> with <i>abc</i> in current line.                      |
| :old/s/old/new          | Find <i>old</i> , then replace <i>old</i> with <i>new</i> .              |
| :1,\$s/old/new/g        | Replace <i>old</i> with <i>new</i> throughout editor buffer.             |





---

## 7 Other Reliant UNIX editors

Reliant UNIX provides several editors for different purposes and editing environments.

The Reliant UNIX editors fall into two basic categories:

- line editors
- full-screen editors

Line editors process only one line at a time, the current line. Full-screen editors, on the other hand, display the contents of a file on the screen; you can modify or add text at any point in the file that is currently visible on the screen.

Important full-screen editors are *vi* and the menu-driven editor *ced*. Important line editors are *ed*, *ex*, *edit* and *sed*. Unlike the other editors mentioned, *sed* is not interactive.

This chapter introduces the following editors:

- *ced* (full-screen, menu-driven editor)
- *ed* (line editor)
- *sed* (noninteractive stream editor)

Each of these three editors is described in a separate section. The description is structured in such a way that each section is complete in itself, enabling you to study each of the three editors separately.

Each section uses a short sample text to demonstrate how the editors work. A good way of familiarizing yourself with the editors is to create a sample text yourself and practise using the functions described.

This chapter gives you a general idea of how each of these three editors works and describes their basic commands. However, it cannot cover the entire functionality of these editors or describe all their commands and how to use them. You will find a detailed description of all the Reliant UNIX editors in the *Commands* manual [5] under the appropriate entries. A detailed description of *vi* is given in chapter 6 *The display editor vi* in this manual.

It is appropriate at this point to mention another Reliant UNIX editor that is not included as a standard function of the operating system. *MAXed* is a powerful, easy-to-use full-screen editor with a wide range of functions. Like *ced*, *MAXed* is menu-driven. For more information, see the *MAXed V3.0B* manual.

## 7.1 ced

*ced* is an easy-to-use, full-screen editor. It is menu-driven and provides you with various editing modes.

You call *ced* using the following standard syntax:

```
ced [file]
```

If you do not specify a *file*, *ced* opens the file you were last editing on your terminal.

### 7.1.1 Creating text

To learn how to use *ced*, you might create a small text file called *test.ced* containing the following text:

```
Baa baa black sheep, have you any wool?
Yes sir, yes sir, three bags full.
One for the master, one for the dame
And one for the little boy who lives down the lane.
```

To do this, you call *ced*:

```
$ ced test.ced
```

The screen will now display a window containing 19 blank lines:

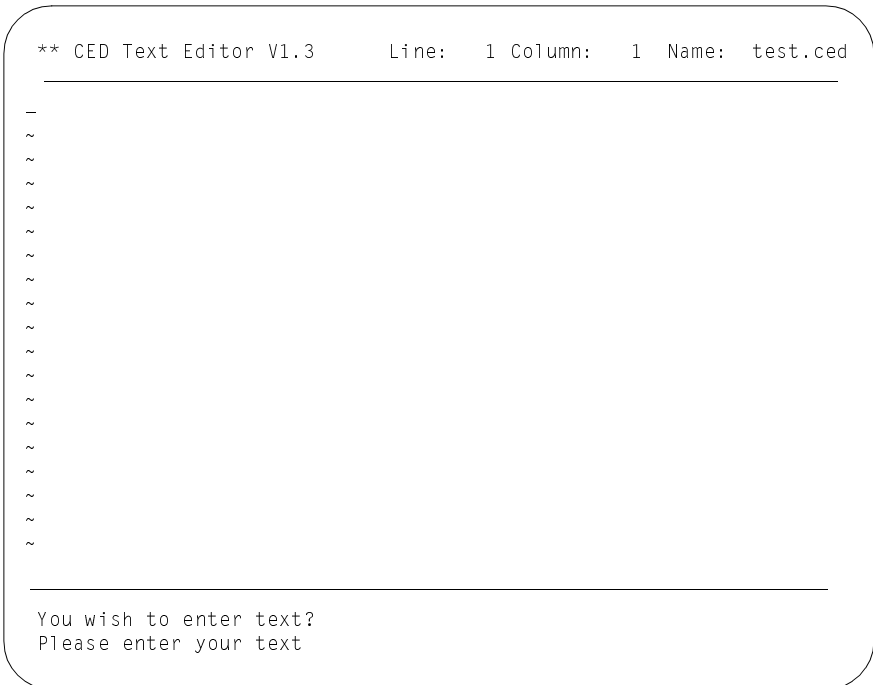


Figure 13: The ced screen

The control panel at the bottom of the window shows you which operating mode you have chosen and which functions are available to you in this mode (detailed information on the *ced* modes is given under *ced* in the *Commands* manual [5] and in section 7.4 *Summary of other Reliant UNIX editors* of this manual).

After you have called *ced*, it is initially in *Enter new text* mode. You can now start to type in your text. If you make a typing error, you can correct it using the backspace key .

Press  when you reach the end of a line to move the cursor to the start of the next line.

Once you have finished typing in your text, it is stored in the editor buffer. The *test.ced* file is still empty. To save the text, you must write the buffer contents to *test.ced* after switching to a different *ced* mode.

### 7.1.2 Saving text

You can use *ced* in various operating modes. The effect of your input depends on the mode you are currently using.

When you call *ced*, it is initially in *Enter new text* mode. If you want to write your text to the *test.ced* file, you must first switch to the *Save text* mode. To switch modes, press **[MENU]**, followed by a letter - in this case, the letter *a* (for *save text*). (For a summary of the *ced* modes, see section 7.4 *Summary of other Reliant UNIX editors*.)

You press the following keys to save *test.ced*:

**[MENU]** *a*

The following text appears in the control panel at the bottom of the screen:

```
You wish to save your text?
Please enter a file name or press <RETURN>
```

If you press **[↓]** instead of entering a file name, *ced* saves the contents of the editor buffer in the file specified when you called *ced*, in this case *test.ced*.

#### Caution:

Note that *ced* overwrites the original version of the file or any existing file of the same name without issuing a warning.

### 7.1.3 Moving the cursor

*ced* recognizes a set of keys that you can use to move the cursor around the screen. The most important of these are described briefly below.

You can use the arrow keys **[↑]** **[↓]** **[→]** **[←]** to move the cursor one character in the direction shown by the arrow. Practice using these keys on your sample text.

**[↓]** and **[↑]** move the screen window 9 lines down or up. You can use **[F15]** and **[F16]** to move the window down or up by one full screen. If you want to move the window to the start of the document, use **[START]**.

### 7.1.4 Deleting and inserting characters

You can use the following keys to correct characters one at a time:

- 

If you press , you move the cursor one character to the left, deleting the character.

- 

If you press , you insert a blank at the cursor position.

- 

If you press , you delete the character on which the cursor is positioned.

Test these keys on your sample text. For example, position the cursor on the start of the word *three* and replace the word with the number 3. To do this, simply enter 3; *ced* then overwrites the letter *t*. To delete the rest of the word, press  four times.

If you now decide that you want to use the word *three* after all, you must first use the

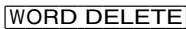
 key to "make room". Then type in *three*.

If you want to insert several characters, however, it is better to switch to the *Insert text* mode (by pressing  i).

### 7.1.5 Deleting and inserting lines

If you press , you delete the line on which the cursor is positioned. To insert a

line in your text, press . The new line is inserted above the line on which your cursor is currently positioned.

 and  have no effect.

## 7.1.6 Quitting ced

To quit *ced*, you switch to the *Quit ced* mode:

```
MENU q
```

The following text appears in the *ced* control panel at the bottom of the editing window:

```
You wish to quit CED?
Save text in document test.ced ?: Enter y=yes or n=no
```

Enter *y* to write the contents of the editor buffer to the specified file.

### Caution:

If you enter *n*, all the work you have done during this *ced* session is lost unless you have saved it in the meantime.

*ced* is then terminated and Reliant UNIX displays the shell prompt.

## 7.2 ed

*ed* is a line editor that can run on any terminal type. For this reason, *ed* is a standard editor in Reliant UNIX, despite its disadvantages in terms of user friendliness.

You call *ed* using the following standard syntax:

```
ed [file]
```

If you do not specify a *file*, *ed* provides you with an empty editor buffer.

### 7.2.1 Creating text in input mode

To obtain an idea of what *ed* can do, you should first create a sample file called *test.ed* with the same contents as the sample file in section 7.1 *ced*:

```
Baa baa black sheep, have you any wool?
Yes sir, yes sir, three bags full.
One for the master, one for the dame
And one for the little boy who lives down the lane.
```

To do this, you call *ed*:

```
$ ed test.ed
```

If the file you specify when calling *ed* does not exist yet, *ed* displays a question mark and the file name.

If you want to edit an existing file and specify its name when calling *ed*, *ed* displays the size of the file in bytes. The file's contents are now loaded in the editor buffer for you to edit (see section 7.2.2 *Editing text in command mode*).

*ed* has two operating modes:

- command mode
- input mode

### Switching to input mode

When you call *ed*, it is initially in command mode. If you want to use *ed* to create text, you must first switch to input mode. *ed* provides you with three commands for this purpose:

- a (append)
- c (change)
- i (insert)

To create a new sample text, enter the *a* command on a separate line and press . Then type in the text, pressing  at the end of each line. Remember that *ed* is a line editor and, as such, can only process one line at a time. If you want to modify a line, but have already pressed  at the end of this line, you must first return to command mode (see section 7.2.2 *Editing text in command mode*).

You use *ed* to create the sample text as follows:

```
$ ed test.ed
?test.ed
a
Baa baa black sheep, have you any wool?
Yes sir, yes sir, three bags full.
One for the master, one for the dame
And one for the little boy who lives down the lane.
.
```

### Exiting input mode

You exit input mode by typing a period (.) on a separate line. The period on the last line of the above example causes *ed* to return to command mode.

The text is now in the editor buffer and you can continue to edit it (for information on saving the contents of the editor buffer, see section 7.2.3 *Saving text*).

### 7.2.2 Editing text in command mode

As already explained, you can edit text using *ed* in command mode. You enter the *ed* commands in the following simple format, which is always the same:

```
[address1[,address2]][command[argument]]
```

You do not actually enter the square brackets. These brackets conform to the command syntax used throughout this guide, and indicate that the entry enclosed within them is optional. You end each *ed* command line by pressing .

The individual components of the command line have the following meaning:

|                          |                                                                                                                                                                                                                                                                                                                   |
|--------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>address1,address2</i> | <i>address1,address2</i> identifies a line block you want to edit using <i>command</i> .<br><i>address1</i> stands for the first line in the block and <i>address2</i> stands for the last line.<br>If you do not specify <i>address2</i> , the command will process only the line specified in <i>address1</i> . |
| <i>command</i>           | <i>command</i> stands for an <i>ed</i> command that instructs the editor to perform a specific function.                                                                                                                                                                                                          |
| <i>argument</i>          | <i>argument</i> tells the <i>ed</i> command what to perform the function on.                                                                                                                                                                                                                                      |

When editing text, make sure that *ed* is in command mode. If you are not sure which operating mode *ed* is currently in, enter a period (.) (the *ed* command for switching to command mode). *ed* is then in command mode.

Some of the *ed* editing functions are described below.

#### Displaying text: the p command

To display a line of text on the screen, you use the *p* command (**print**).

This command has the following standard syntax:

```
[address1[address2]]p
```

You can use the *p* command to display any line of text. For example, you display the second line of text in the sample file by entering

```
2p
Yes sir, yes sir, three bags full.
```



If you enter *p* without specifying a line number, the current line, i.e. the line you edited last, is displayed.

You can also use line numbers to specify a line block. To do this, enter the address of the first and last line of the block you want to display, separated by commas. You can use the dollar sign (\$) as the symbolic address of the last line (other symbolic addresses are listed under *ed* in the *Commands* manual [5]).

The following example illustrates how to display the first three lines of the sample file, followed by the block from line 3 to the end of the file:

```
1,3p
Baa baa black sheep, have you any wool?
Yes sir, yes sir, three bags full.
 One for the master, one for the dame

3,$p
One for the master, one for the dame
And one for the little boy who lives down the lane.
```

### Substituting text: the *s* command

To substitute one string with another, you use the *s* command.

You can use this command to substitute a string within a line or block of lines.

You specify the line number or block (address of the first and last line in the section of text involved) in front of the *s* command.

This command has the following standard syntax:

```
address1[,address2]s/pattern/replacement_string/[parameter]
```

*ed* allows you to specify a regular expression as the *pattern*. For detailed information, see *ed* in the *Commands* manual [5].

*ed* replaces the first occurrence of *pattern* on a line. If *pattern* occurs more than once on the same line, *ed* replaces only the first occurrence. If you want to replace every occurrence of a pattern in a text, *ed* allows you to specify the *g* parameter for global substitution. For detailed information, see *ed* in the *Commands* manual [5]).

The following example illustrates how to replace the string *three* in line 2 with the number 3:

```
2s/three/3
Yes sir, yes sir, 3 bags full.
```

### Deleting a line: the **d** command

To delete a line of text, you use the *d* command.

You can use this command to delete a single line or a block of lines.

The command has the following standard syntax:

```
[address1[address2]]d
```

The following example shows how you delete lines 3 through 4:

```
3,4d
```

### Undoing the previous command: the **u** command

You use the *u* command to undo the previous command.

The text that you modified or deleted using the last command is restored completely when you enter the *u* command.

## 7.2.3 Saving text

During an *ed* session, your text is stored in the editor buffer. If you want to save the text, you must write the buffer contents to a file.

To write the buffer contents to a file, you use the *w* command.

This command has the following standard syntax:

```
w [file]
```

If you do not specify a file, *ed* saves the buffer contents to the file specified when you called the editor. In the above example, therefore, the *w* command would write the buffer contents to *test.ed*.

## 7.2.4 Quitting ed

To quit *ed*, you use the *q* command. *ed* is then terminated and Reliant UNIX displays the shell prompt.

### Caution:

Make sure that you write the buffer contents to a file before quitting *ed* (see section 7.2.3 *Saving text*). If you enter the *q* command without first saving the buffer contents using *w*, *ed* displays a question mark (?) as a warning. If you then repeat the *q* command immediately, you quit *ed*, destroying the buffer contents.

## 7.2.5 Using ed scripts

*ed* reads its commands and text from standard input. By redirecting standard input, you can instruct *ed* to read from a file. For detailed information on redirecting standard input, see chapter 5 *The shell*. This gives you the option of writing *ed* commands to a file and passing them to *ed* as required, thus enabling you to reproduce specific command sequences at any time.

In the following example, the *ed* commands in a file called *ed.script* are passed to the editor. *ed.script* has the following contents:

```
2s/three/3
w
```

In the first line, the *ed* command used in the section on *Searching for and replacing text* is passed to *ed*. This command replaces the character string *three* in the second line of *test.ed* with the number 3. The *ed* command *w* then saves the modified text to the file specified when you called *ed*, in this case *test.ed*.

The following command line instructs *ed* to edit *test.ed* using the commands specified in the *ed.script* file:

```
$ ed test.ed < ed.script
```

## 7.3 sed

*sed* is a noninteractive stream editor that provides a similar set of functions to *ed*.

Since *sed* is *noninteractive*, it is unable to "communicate" with the user. When you call *sed*, it processes the *sed* commands passed to it on the command line. *sed* then terminates and Reliant UNIX displays the shell prompt.

*sed* performs the following operations:

- It reads the lines of a file sequentially into the editor buffer.
- It edits the line in the editor buffer using the *sed* commands you have specified.
- It writes the contents of the editor buffer to standard output and deletes the editor buffer.

Because *sed* is a noninteractive editor, it is particularly useful for processing sequences of editing commands in one pass or editing piped command output (for detailed information on redirecting standard output using a pipeline, see chapter 5 *The shell*).

*sed* reads the text editing commands either from the command line or from a file (*sed* script).

When *sed* reads from the command line, it has the following standard syntax:

```
sed -e command(s) [file]
```

If you specify more than one *sed* command, terminate each one using .

If you do not specify a *file*, *sed* reads from standard input. The fact that *sed* can read from standard input is of great importance, since this enables it to edit the output of other Reliant UNIX commands via a pipeline (for information on editing the output of Reliant UNIX commands using *sed*, see section 7.3.4 *Editing text in a pipeline*; for information on redirecting standard output using a pipeline, see chapter 5 *The shell*).

If you use a *sed* script, *sed* has the following standard syntax:

```
sed -f script_name [file]
```

The next section describes the format of a *sed* script.

### 7.3.1 Specifying sed commands

You specify *sed* commands in the following format:

```
[address1[address2]]command[argument]
```

or, if your command input is more than one line long:

```
[address1[address2]]{command[argument]
 command[argument]
 .
 .
 .
}
```

You do not actually type the square brackets. These conform to the command syntax used throughout this guide and indicate that the entries enclosed in them are optional.

You do not have to type the braces unless you have specified *address1* or *address1* and *address2*.

The right brace (*}*) must appear at the start of the line, i.e. only blanks or tabs may appear to the left of it.

If you do not specify a line block for *address*, the *command* is executed for all the lines.

|                          |                                                                                                                                                                                                                                                                                                                   |
|--------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>address1,address2</i> | <i>address1,address2</i> identifies a line block you want to edit using <i>command</i> .<br><i>address1</i> stands for the first line in the block and <i>address2</i> stands for the last line.<br>If you do not specify <i>address2</i> , the command will process only the line specified in <i>address1</i> . |
| <i>command</i>           | <i>command</i> stands for a <i>sed</i> command that instructs the editor to perform a specific function.                                                                                                                                                                                                          |
| <i>argument</i>          | <i>argument</i> tells the <i>sed</i> command what to perform the function on.                                                                                                                                                                                                                                     |

As already explained, you can pass commands to *sed* either on the command line or in an *sed* script.

If your *sed* statement contains blanks or tabs, you must enclose it in single quotes (*'*) when you pass it to *sed* on the Reliant UNIX command line.

### 7.3.2 Editing text

Try the following examples to see how *sed* works. First create a sample file called *test.sed* with the same contents as the sample file in the two previous sections:

```
Baa baa black sheep, have you any wool?
Yes sir, yes sir, three bags full.
One for the master, one for the dame
And one for the little boy who lives down the lane.
```

It is advisable to create this file using a different editor or the *cat* command (see chapter 3 *The file system*). *sed* is suitable for editing text, but not creating it.

Familiarize yourself with *sed* by trying the following examples yourself.

#### Deleting lines: the *d* command

To delete a section of text, you use the *d* command.

You can use this command to delete a single line or a block of lines.

The command has the following standard syntax:

```
[address1[address2]]d
```

If you do not specify a line block for *address*, this command deletes all the lines.

The following example illustrates how to delete lines 3 through 4 of the sample file:

```
$ sed -e 3,4d test.sed
Baa baa black sheep, have you any wool?
Yes sir, yes sir, three bags full.
$
```

You could also pass this *sed* command to the editor as a *sed* script containing the following line:

```
3,4d
```

You then call *sed* as follows:

```
$ sed -f script test.sed
Baa baa black sheep, have you any wool?
Yes sir, yes sir, three bags full.
$
```

*sed* loads all the lines in the *test.sed* file into the editor buffer. It then edits the lines in the specified block using the commands specified on the command line or in the *sed* script. Once *sed* has edited the lines loaded into the editor buffer, it writes them to standard output.

In the above example, for instance, *sed* loads the first and second lines into the buffer. However, since these lines are outside the specified line block, *sed* does not edit them, but writes them unchanged to standard output. The third and fourth lines, on the other hand, are loaded into the editor buffer and edited, in this case deleted, and are no longer written to standard output.

## Inserting text

To insert text, you use the *a* (append) or the *i* command (insert).

These commands have the following standard syntax:

```
[address]a\
text
```

and

```
[address]i\
text
```

If you use the *a* command, you insert *text* after the line specified in *address*.

If you use the *i* command, you insert *text* before the line specified in *address*.

The command in the following example inserts the text (*Nursery rhyme*) below line 4. Because the text contains blanks, you must enclose the *sed* command in single quotes (').

```
$ sed -e '4a\
> (Nursery rhyme)' test.sed
Baa baa black sheep, have you any wool?
Yes sir, yes sir, three bags full.
One for the master, one for the dame
And one for the little boy who lives down the lane.
(Nursery rhyme)
$
```

*sed* loads all the lines into the editor buffer and then writes them to standard output, appending the text (*Nursery rhyme*) below line 4.

## Substituting text

To substitute one string with another, you use the *s* command.

You can use this command to substitute a string within a line or block of lines.

You specify the line number or block (address of the first and last line in the section you want to replace) in front of the *s* command.

The command has the following standard syntax:

```
[address1[,address2]]s/pattern/substitute_string/[parameter]
```

If you do not specify a line block for *address*, the command is executed for all the lines.

*sed* allows you to specify a regular expression as the *pattern*. For detailed information on regular expressions, see the *Appendix* of this manual.

*sed* replaces the first occurrence of *pattern* on a line. If *pattern* occurs more than once on the same line, *sed* replaces only the first occurrence. To replace a pattern throughout the entire text, you can specify the *g* parameter for global string substitution. Further information on this parameter is given at the end of this section. For more detailed information, see *ed* in the *Commands* manual [5].

In the following example, the string *three bags* is replaced with *3 sacks*. Because the *pattern* and *substitute\_string* contain blanks, you must enclose the *sed* command in single quotes (').

```
$ sed -e s/'three bags'/'3 sacks'/
Baa baa black sheep, have you any wool?
Yes sir, yes sir, 3 sacks full.
One for the master, one for the dame
And one for the little boy who lives down the lane.
$
```

*sed* loads the lines into the editor buffer, edits them and then writes them to standard output. The command finds the pattern *three bags* in the second line and replaces it with the substitute string *3 sacks*.

## Editing text using an *sed* script

The *sed* command you used in the section on *Substituting text* could also be passed to the editor as a *sed* script containing the following line:

```
s/three bags/3 sacks/
```

The next example adds two more commands to the *sed* script *script*, which then contains three *sed* commands separated by blank lines:

```
s/three bags/3 sacks/
4a\
End of nursery rhyme.
2,3s/^/ /
```



*script* has the following effect:

- The string *three bags* is replaced with *3 sacks*.
- The line *End of nursery rhyme.* is appended after the fourth line (4a).
- *sed* looks for the start of the line on lines 2 and 3 (^ is the regular expression that represents the start of the line) and replaces it with four blanks. In other words, lines 2 and 3 are indented by four characters.

You then call *sed* as follows:

```
$ sed -f script test.sed
Baa baa black sheep, have you any wool?
Yes sir, yes sir, 3 sacks full.
One for the master, one for the dame
And one for the little boy who lives down the lane.
End of nursery rhyme.
$
```

The next example demonstrates *sed*'s global substitution function. You want to replace every occurrence of the letter *s* with the letter *x*. To replace a *pattern* globally in a text, specify the *g* parameter after the substitute string (in this case the letter *x*) in the *sed* command *s*:

```
$ sed -e s/s/x/g test.sed
Baa baa black xheep, have you any wool?
Yex xir, yex xir, three bagx full.
One for the maxter, one for the dame
And one for the little boy who livex down the lane.
$
```

### 7.3.3 Saving text

*sed* writes the lines loaded into the editor buffer to standard output, not to a file.

If you want to write the text you have edited using *sed* to a file, you can:

- use the *w* (write) command
- redirect standard output to a file

#### Writing the contents of the editor buffer to a file: the *w* command

The *w* command writes the contents of the editor buffer to a file.

This command has the following standard syntax:

```
[address1[,address2]]w sed.file
```

Note that *w* only saves text that has first been loaded into the editor buffer. For example, text inserted using the *a* or *i* commands is written to standard output, but not loaded into the editor buffer. For this reason, it is not written to *sed.file*.

**Caution:**

If there is already a file called *sed.file*, it is overwritten without warning. If you use the name of your input file for *sed.file*, you will destroy your input file.

In the following example, the third and fourth lines of the sample file are deleted and the result written to a file called *sed.out*.

```
$ sed -e '3,4d
> w sed.out' test.sed
Baa baa black sheep, have you any wool?
Yes sir, yes sir, three bags full.
$ cat sed.out
Baa baa black sheep, have you any wool?
Yes sir, yes sir, three bags full.
$
```

**Redirecting standard output to a file**

By redirecting standard output, you can write the *sed* output to a file.

This takes advantage of the fact that *sed* writes all the lines that are loaded into the editor buffer, together with all the lines that have been added to the text, to standard output.

You use the following standard syntax for this purpose:

```
sed -e command input_file > sed.file
```

**Caution:**

If there is already a file called *sed.file*, it is overwritten without warning. If you use the name of your input file as *sed.file*, you will destroy your input file.

The next example uses the *a* command to append the text (*Nursery rhyme*) after the fourth line, and redirects standard output to a file called *sed.out*.

```
$ sed -e '4a\
> (Nursery rhyme)' test.sed > sed.out
$ cat sed.out
Baa baa black sheep, have you any wool?
Yes sir, yes sir, three bags full.
One for the master, one for the dame
And one for the little boy who lives down the lane.
(Nursery rhyme)
$
```

### 7.3.4 Editing text in a pipeline

Because *sed* can read from standard output, it can edit the output of other Reliant UNIX commands via a pipeline. This section describes this feature of *sed*.

Detailed information on redirecting the output of a Reliant UNIX command using a pipeline is given in chapter 5 *The shell*.

The following example uses *sed* to edit the output of the *who* command (for determining which users are logged in). *who* outputs the following information:

```
$ who
darkstar tty002 Feb 10 09:12
arthur2 tty004 Feb 10 08:06
james tty007 Feb 10 11:55
tsv tty018 Feb 10 06:00
$
```

Since you want to replace the string *darkstar* with the string *xxxxxxx*, enter the following command line:

```
$ who | sed -e s/darkstar/xxxxxxx/
xxxxxxx tty002 Feb 10 09:12
arthur2 tty004 Feb 10 08:06
james tty007 Feb 10 11:55
tsv tty018 Feb 10 06:00
$
```

## 7.4 Summary of other Reliant UNIX editors

| ced summary                |                                                   |
|----------------------------|---------------------------------------------------|
| ced modes                  |                                                   |
| Key                        | Mode                                              |
| <b>[MENU]</b> a            | Save file                                         |
| <b>[MENU]</b> b            | Edit line <b>b</b> lock                           |
| <b>[MENU]</b> c            | Execute Reliant UNIX <b>c</b> ommand              |
| <b>[MENU]</b> d            | Switch <b>d</b> ocuments                          |
| <b>[MENU]</b> e            | <b>E</b> nter new text                            |
| <b>[MENU]</b> i            | Insert text                                       |
| <b>[MENU]</b> l            | Edit line                                         |
| <b>[MENU]</b> p            | <b>P</b> rogram keys                              |
| <b>[MENU]</b> q            | <b>Q</b> uit ced                                  |
| <b>[MENU]</b> r            | Edit <b>r</b> ectangle                            |
| <b>[MENU]</b> s            | <b>S</b> earch for string                         |
| <b>[MENU]</b> t            | Find <b>t</b> ag                                  |
| <b>[MENU]</b> w            | Position <b>w</b> indow                           |
| Calling ced                |                                                   |
| ced <b>[↓]</b>             | Load <i>ced</i> with the <i>file</i> to be edited |
| ced <i>file</i> <b>[↓]</b> | Load <i>ced</i> with the <i>file</i> to be edited |
| Quitting ced               |                                                   |
| <b>[MENU]</b> qy           | Quit <i>ced</i> and save changes in current file  |
| <b>[MENU]</b> qn           | Quit <i>ced</i> without saving changes            |

| Positioning the cursor                                                                                                                                                  |                                                    |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------|
|                                                                                        | Start of file                                      |
|       | First line of file                                 |
|       | Last line of file                                  |
|                                                                                        | 1/2 window up                                      |
|                                                                                        | 1/2 window down                                    |
|                                                                                        | 1 window down                                      |
|                                                                                        | 1 window up                                        |
|                                                                                        | One line up                                        |
|       | First line of window                               |
|       | First line of file                                 |
|                                                                                        | One line down                                      |
|       | Last line of window                                |
|       | Last line of file                                  |
|                                                                                        | One character to the left                          |
|                                                                                        | Tab left (tab stops are in columns 1, 9, 17 etc.)  |
|       | Left-hand edge of window                           |
|       | Start of line                                      |
|       | Start of line                                      |
|                                                                                      | One character to the right                         |
|                                                                                      | Tab right (tab stops are in columns 1, 9, 17 etc.) |
|   | Right-hand edge of window                          |
|   | End of line                                        |
|   | End of line                                        |

|                                   |                                            |
|-----------------------------------|--------------------------------------------|
| Inserting and deleting characters |                                            |
| CHAR INSERT                       | Insert character                           |
| CHAR DELETE                       | Delete character                           |
| ⌫                                 | Backspace; delete character left of cursor |
| Inserting and deleting lines      |                                            |
| LINE INSERT                       | Insert line                                |
| LINE DELETE                       | Delete line                                |
| Other commands                    |                                            |
| MENU !                            | Call shell                                 |
| MENU p                            | Program keys                               |
| MENU x                            | Show programmed keys                       |

Table 93: ced summary

|                                                  |                                                                                                             |
|--------------------------------------------------|-------------------------------------------------------------------------------------------------------------|
| <b>ed summary</b>                                |                                                                                                             |
| <b>Activating input mode</b>                     |                                                                                                             |
| a<br>c<br>i                                      | Append text<br>Change text<br>Insert text                                                                   |
| <b>Switching from input mode to command mode</b> |                                                                                                             |
| .                                                |                                                                                                             |
| <b>ed commands (selection)</b>                   |                                                                                                             |
| d                                                | Delete line                                                                                                 |
| h                                                | Explain last error message ( <b>h</b> - help)                                                               |
| H                                                | Toggle <b>help</b> mode on and off<br>If help mode is on, error messages are printed for all ? diagnostics. |
| n                                                | Number and print lines                                                                                      |
| p                                                | Print line                                                                                                  |
| P                                                | Use * as prompt                                                                                             |
| q                                                | Quit <i>ed</i>                                                                                              |
| Q                                                | Quit <i>ed</i> ; <i>ed</i> does not issue a warning if you have not saved the buffer contents.              |
| s                                                | Search for and replace string ( <b>s</b> - substitute)                                                      |
| u                                                | Undo last command                                                                                           |
| w                                                | Write buffer contents to file                                                                               |
| W                                                | Append buffer contents to file ( <b>W</b> - write)                                                          |
| !                                                | Send command to shell for interpretation                                                                    |
| [DEL]                                            | Abort execution of command                                                                                  |

Table 94: ed summary

| sed summary              |                                                               |
|--------------------------|---------------------------------------------------------------|
| Format of command input  |                                                               |
| -e <i>command(s)</i>     | <i>sed</i> reads the commands you specify on the command line |
| -f <i>script</i>         | <i>sed</i> reads commands from a <i>sed</i> script            |
| sed commands (selection) |                                                               |
| a                        | <b>A</b> ppend line                                           |
| d                        | <b>D</b> elete line                                           |
| i                        | <b>I</b> nsert line                                           |
| q                        | <b>Q</b> uit <i>sed</i>                                       |
| s                        | Search for and replace string ( <b>s</b> - substitute)        |
| w                        | <b>W</b> rite buffer contents to file                         |

Table 95: sed summary



---

# 8 The awk programming language

## 8.1 Introduction

*awk* is a programming language similar to C. It is particularly suitable for text editing tasks. For example, you can use it to create forms, search for strings in files, modify the format of input data or perform arithmetic on the data in a file.

The *awk* programming language (its name is an acronym formed from the initials of its developers Aho, Weinberger and Kernighan) includes the following features:

- *awk* processes one record at a time. An input record is assumed to be a line, unless the user redefines this default setting.
- Each input record consists of fields that can be addressed individually.
- Since it is an interpreter, *awk* is particularly useful for step-by-step problem solving.
- *awk* provides you with many functions that enable you to solve even complex problems using a small number of statements. For this reason, *awk* programs are usually relatively short.

To learn *awk*, you need a knowledge of Reliant UNIX and shell programming. Although a knowledge of the C programming language is not essential, it is useful because of the similarity between the two languages.

**Note:**

Once you have learned the basics of the *awk* programming language, you should read section 8.9.1 *Some useful one-liners*. This section includes some simple solutions to frequent programming problems.

## 8.2 awk program structure

An *awk* program can consist of three parts: a *BEGIN* pattern, a main section containing one or more pattern-action statements and an *END* pattern. It must contain at least one of these three parts, whether you enter the *awk* program on the command line or create a file for it. An *awk* program has the following structure:

### awk program structure

```
{BEGIN action}

 {pattern 1 action}
 { . . }
 { . }
 {pattern n action}

{END action}
```

The following example illustrates what a coded *awk* program looks like:

```
BEGIN { print "Start of output" }
$4 == "Asia" { print $1, $2 }
END { print "End of output" }
```

The above example is a typical *awk* program, containing the optional *BEGIN* and *END* pattern. This program contains one pattern-action statement between the *BEGIN* and *END* pattern.

You could include these statements in an *awk* program file or specify them on the *awk* command line, in which case you would enclose them in single quotes (the next section, *8.3 Calling awk*, explains how to start this program). The program in this example first writes the string *Start of output* to standard output. It then outputs the first and second field (*\$1* and *\$2*) of each line of the input file in which the fourth field contains the entry *Asia*. Finally, it outputs the string *End of output*.

*awk* programs generally execute as follows: if there is a *BEGIN* pattern, *awk* first executes the action in the *BEGIN* pattern. The program then processes the specified pattern-action statements by comparing each line of the input file with each specified pattern. If a pattern matches an input line, the action associated with that pattern is executed. Then the next input line is processed in the same way. Finally, if there is an *END* pattern, the action associated with the *END* pattern is executed.

Note that you may omit either the pattern or the action in a pattern-action statement. If no action is specified for a pattern, as in

```
$1 == "name"
```

*awk* simply writes the matching line to standard output. If, however, no pattern is specified for an action, as in

```
{ print $1, $2 }
```

*awk* executes the action for every input line. You must enclose actions in braces (*{}*) to distinguish them from patterns.

## 8.3 Calling awk

There are two ways of calling an *awk* program. You either enter the program on the command line or include the statements in a file and instruct *awk* to process the statements in this file. Files are particularly useful if you want *awk* to process more than one statement.

### Calling awk from the command line

You can call an *awk* program directly from the command line using the following general format:

```
awk 'BEGIN pattern
 pattern-action statement(s)
 END pattern' optional list of input files
```

The following example does not include the optional *BEGIN* and *END* patterns. The pattern used in the example is the expression *\$4 == "Asia"*. The action associated with this pattern is the *print* statement enclosed in braces (*{}*). This short program reads in each record of the input file *file* and outputs the first and second field if the fourth field contains the string *Asia*. You enter the following command line:

```
$ awk '$4 == "Asia" { print $1, $2 }' file
```

The action is enclosed in single quotes (') to prevent the shell from interpreting the \$ as a metacharacter. Instead of using single quotes, you could of course escape each character individually using the backslash (\). It is easier and less error-prone, however, to use single quotes. If you do not specify a file name on the command line, *awk* expects to receive its input data from standard input. You can also instruct *awk* explicitly to read from standard input, however, by specifying the hyphen (-) as one of the input files. For example, the following command tells *awk* to read its input first from *file* and then from standard input:

```
$ awk '{ print $3, $4 }' file -
```

### Calling awk using a program file

Calling *awk* from the command line with the pattern-action statements enclosed in single quotes is convenient above all when the *awk* program is only a few lines long. If you want to write a long program, however, it is advisable to include the statements in a file. This makes it easier to correct the *awk* program, since you do not have to reenter it completely in the event of a syntax error. You call this file with the -f option, using the following general format:

```
awk -f program_file optional list of input files
```

The following command line calls the *myprogram* program with *file* as the input file:

```
$ awk -f myprogram file
```

## 8.4 Patterns

In a pattern-action statement, a pattern is a criterion for selecting the records for which the associated action is executed. *awk* recognizes three types of pattern:

- The first type of pattern is the relational expression. You use relational expressions to make comparisons.
- The second type of pattern is the regular expression. Regular expressions enable you select records by means of pattern matching.
- The third type of pattern is the optional *BEGIN* and *END*. *BEGIN* and *END* are two special patterns in an *awk* program.

The following examples often use a file called *countries* as the input file. This file contains information on the ten largest countries in the world. Each record contains the name of a country, its area in thousands of square miles, its population in millions and the continent on which it is located (the data is from

1978; what was known then as the USSR has been placed arbitrarily in Asia). The white space between the fields is a tab in the original input; a single blank separates *North* and *South* from *America*.

The sample file *countries* looks like this:

|           |      |     |               |
|-----------|------|-----|---------------|
| USSR      | 8650 | 262 | Asia          |
| Canada    | 3852 | 24  | North America |
| China     | 3692 | 866 | Asia          |
| USA       | 3615 | 219 | North America |
| Brazil    | 3286 | 116 | South America |
| Australia | 2968 | 14  | Australia     |
| India     | 1269 | 637 | Asia          |
| Argentina | 1072 | 26  | South America |
| Sudan     | 968  | 19  | Africa        |
| Algeria   | 920  | 18  | Africa        |

This file is typical of the kind of data *awk* is good at processing - a mixture of words and numbers separated into fields by blanks and tabs. For detailed information on working with an input file, see section 8.6 *Input*.

### 8.4.1 The first type of pattern: relational expressions

The first type of pattern, the relational expression, allows you to make comparisons. For example, the operator `==` checks whether the operands are equal. Suppose that you want to output all the lines of the input file *countries* in which the fourth field matches the string *Asia*. To do this, you use a simple program that consists only of a single pattern and an associated action:

```
$ awk '$4 == "Asia" { print }' countries
```

This program produces the following output:

|       |      |     |      |
|-------|------|-----|------|
| USSR  | 8650 | 262 | Asia |
| China | 3692 | 866 | Asia |
| India | 1269 | 637 | Asia |

*awk* supports the following relational operators:

`>`, `>=`, `<`, `<=`, `==` (equal to) and `!=` (not equal to).

You can use these relational operators to compare both numbers and strings. Let us assume that you want to restrict the output to countries with a population of over 100 million. You can do so using the following program:

```
$ awk '$3 > 100 { print }' countries
```

This program outputs all the lines in which the third field contains a value greater than 100. You will recall that the third field in the *countries* file contains the population in millions.

*awk* also recognizes the two relational operators `~` (tilde) and `!~` (exclamation mark and tilde). Table 96 lists these operators and their functions.

| Operator           | Function                 |
|--------------------|--------------------------|
| <code>&lt;</code>  | less than                |
| <code>&lt;=</code> | less than or equal to    |
| <code>==</code>    | equal to                 |
| <code>!=</code>    | not equal to             |
| <code>&gt;=</code> | greater than or equal to |
| <code>&gt;</code>  | greater than             |
| <code>~</code>     | matches                  |
| <code>!~</code>    | does not match           |

Table 96: *awk* relational operators

If both operands are numeric, a numeric comparison is made; otherwise the operands are treated as strings and an alphanumeric comparison is made. Both operands can be either a number or a character string; *awk* normally determines the data type automatically. For further information, see the subsection *Number or string*. For example, the pattern `$3>100` selects lines in which the value of the third field is greater than *100*. The program

```
$ awk '$1 >= "S" { print }' countries
```

selects lines that start with a letter between *S* and *Z*:

```
USSR 8650 262 Asia
USA 3615 219 North America
Sudan 968 19 Africa
```

In the absence of any other information, *awk* treats a field as a character string. The program

```
$ awk '$1 == $4 { print }' countries
```

makes a string comparison between the first and fourth field. Only one line in the input file *countries* meets this condition:

```
Australia 2968 14 Australia
```

If both fields are obviously numeric, the program makes a numeric comparison.

## 8.4.2 The second type of pattern: regular expressions

The options offered by *awk* for searching for strings are not limited to the comparisons described in the last section. You can perform more complex searches using powerful patterns known as regular expressions. The regular expressions in *awk* have the same format as those in *egrep* and *lex*. The simplest regular expression is a string enclosed in slashes, e.g.

```
/US/
```

The program outputs each line that contains the substring *US*:

```
$ awk '/US/ { print $0 }' countries
USSR 8650 262 Asia
USA 3615 219 North America
$
```

If *re* stands for a regular expression, the pattern

```
/re/
```

matches any line that contains the substring you specify in the regular expression *re*.

You can also restrict the match to a specific field using the pattern matching operators *~* (matches) and *!~* (does not match). The program

```
$ awk '$4 ~ /Asia/ { print $1 }' countries
```

outputs the first field of all lines in which the fourth field matches *Asia*, while the program

```
$ awk '$4 !~ /Asia/ { print $1 }' countries
```

outputs the first field of all lines in which the fourth field does *not* match *Asia*.

Regular expressions can contain the characters

```
\ ^ $. [] * + ? () |
```

that have the same special meaning as the shell metacharacters. For example, the circumflex (^) and the dollar sign (\$) match the start and end of a string respectively, and the period (.) matches any single character. The regular expression

```
/^,$/
```

thus matches all records that contain exactly one character.

A group of characters enclosed in square brackets matches any one of the enclosed characters. For example, `/[ABC]/` matches all records that contain one of the characters *A*, *B* or *C*. Ranges of letters or numbers can be abbreviated within square brackets: `/[a-zA-Z]/` matches any single letter.

If the first character to the right of the square bracket (*I*) is a circumflex (`^`), *awk* searches for the *complement* to the following characters, i.e. only those characters that are *not* specified within the brackets. `/[^a-zA-Z]/` thus matches any character that is not a letter. The plus sign `+` stands for one or more occurrences. The program

```
$ awk '$2 !~ /^[0-9]+$/' {print }' countries
```

thus outputs all records in which the third field does not contain a string of one or more numeric characters (`^` stands for the start of the string, `[0-9]+` stands for one or more numeric characters and `$` stands for the end of the string).

Regular expressions can also contain parentheses `()` for grouping elements and the logical OR `()` for specifying alternatives. The program

```
$ awk '/(apple|cherry) (pie|tart)/ { print }' file
```

searches for all lines that contain one of the four substrings *apple pie*, *apple tart*, *cherry pie* and *cherry tart*.

You can escape a metacharacter by prefixing it with a backslash (`\`). In addition to various metacharacters, *awk* recognizes the following C programming language escape sequences within regular expressions and strings:

| Escape sequences  | Meaning                                                                                               |
|-------------------|-------------------------------------------------------------------------------------------------------|
| <code>\b</code>   | The backspace key  |
| <code>\f</code>   | Form feed                                                                                             |
| <code>\n</code>   | Newline                                                                                               |
| <code>\r</code>   | The  key           |
| <code>\t</code>   | Tab                                                                                                   |
| <code>\ddd</code> | Octal value <i>ddd</i>                                                                                |
| <code>\"</code>   | Quotation mark                                                                                        |
| <code>\c</code>   | Any other literal character <i>c</i>                                                                  |

For example, you could use the following pattern to display all lines containing a tab:

```
/\t/
```



awk interprets any string or variable to the right of a tilde (~) or an exclamation mark and tilde (!~) as a regular expression. For example, you could also have entered the statement

```
$2 !~ /^[0-9]+$ /
```

as:

```
BEGIN { digits = "[0-9]$" }
$2 !~ digits
```

Let us assume that you want to search for a string such as `^[0-9]+$`. If you specify a literal quoted string like `"^[0-9]+$"` as a regular expression, you must include additional backslashes (`\`) to prevent the metacharacter in the regular expression from being interpreted by the shell. This is because the first set of backslashes is removed when a string is first parsed. If you need to use a backslash to escape a metacharacter in a regular expression, you must also protect the backslash itself by prefixing it with another backslash.

You now want to search for strings that contain a lowercase *b* followed by a dollar sign (`$`). The regular expression for this pattern is: `b\$`. The string for this regular expression must include an additional backslash: `"b\$"`. The following table lists pairs of regular expressions that have the same function:

|                           |        |                          |
|---------------------------|--------|--------------------------|
| <code>x ~ "b\\ \$"</code> | equals | <code>x ~ /b\ \$/</code> |
| <code>x ~ "b\ \$"</code>  | equals | <code>x ~ /b\$/</code>   |
| <code>x ~ "b\$"</code>    | equals | <code>x ~ /b\$/</code>   |
| <code>x ~ "\ t"</code>    | equals | <code>x ~ /\ t/</code>   |

Table 97 shows the exact format of the regular expressions and the substrings they match. The unary operators `*`, `+` and `?` have the highest priority, followed by concatenation, then alternation (`|`). The operators are parsed from left to right; *r* stands for any regular expression.

|                    |                                                                |
|--------------------|----------------------------------------------------------------|
| <code>c</code>     | Character <i>c</i> ; <i>c</i> may not be a wildcard character. |
| <code>\c</code>    | Literal character <i>c</i> .                                   |
| <code>^</code>     | Start of string.                                               |
| <code>\$</code>    | End of string.                                                 |
| <code>.</code>     | Any character except newline.                                  |
| <code>[s]</code>   | Any character in set <i>s</i> .                                |
| <code>[^s]</code>  | Any character not in set <i>s</i> .                            |
| <code>r*</code>    | Zero or more regular expressions <i>r</i> .                    |
| <code>r+</code>    | One or more regular expressions <i>r</i> .                     |
| <code>r?</code>    | Zero or one regular expression <i>r</i> .                      |
| <code>(r)</code>   | <i>r</i>                                                       |
| <code>r1 r2</code> | <i>r1</i> followed by <i>r2</i> (concatenation).               |
| <code>r1 r2</code> | <i>r1</i> or <i>r2</i> (alternation).                          |

Table 97: awk regular expressions

### 8.4.3 The third type of pattern: BEGIN and END

*BEGIN* and *END* are two special patterns. They allow you to specify what actions you want to take place at the start and end of program execution. The action associated with *BEGIN* is executed before *awk* reads in the first input record. The *END* pattern matches the end of input, after the last input record has been processed.

In the following program, the *BEGIN* pattern outputs the title *Countries of Asia* before any input records are read in:

```
$ awk 'BEGIN { print "Countries of Asia:" }
> /Asia/ { print " ", $1 }' countries
Countries of Asia:
USSR
China
India
$
```

The following *awk* program uses the *BEGIN* pattern to set the field separator to tab (`\t`) and add column headings to the output. The field separator is stored in the special variable *FS*. Although you can reset *FS* at any time, it makes sense to do so in the *BEGIN* pattern, i.e. before the first input line is read in. The second *printf* statement in the program, which is executed for each input line,

formats the output into a table, neatly aligned under the column headings. The *END* action outputs the totals. Because this program is relatively complex, it is advisable to write it to a file. Note that a long line can be continued after a comma.

```
BEGIN { FS = "\t"
 printf "%10s %6s %5s %s\n",
 "COUNTRY", "AREA", "POP", "CONTINENT" }
{ printf "%10s %6d %5d %s\n", $1, $2, $3, $4
 area = area + $2; pop = pop + $3 }
END { printf "\n%10s %6d %5d\n", "TOTAL", area, pop }
```

With the *countries* file as input, this program produces the following output:

| COUNTRY   | AREA  | POP  | CONTINENT     |
|-----------|-------|------|---------------|
| USSR      | 8650  | 262  | Asia          |
| Canada    | 3852  | 24   | North America |
| China     | 3692  | 866  | Asia          |
| USA       | 3615  | 219  | North America |
| Brazil    | 3286  | 116  | South America |
| Australia | 2968  | 14   | Australia     |
| India     | 1269  | 637  | Asia          |
| Argentina | 1072  | 26   | South America |
| Sudan     | 968   | 19   | Africa        |
| Algeria   | 920   | 18   | Africa        |
| TOTAL     | 30292 | 2201 |               |

## 8.4.4 Compound patterns

A compound pattern combines simpler patterns with parentheses and the logical operators `||` (or), `&&` (and) and `!` (not). Let us assume that you want to display a list of all the countries in Asia with a population of more than 500 million. The following program does this by selecting all lines in which the fourth field contains the string *Asia* and the third field contains a value greater than 500:

```
$ awk '$4 == "Asia" && $3 > 500 { print }' countries
```

The program

```
$ awk '$4 == "Asia" || $4 == "Africa" { print }' countries
```

searches for all lines in which the fourth field contains the string *Asia* or *Africa*. Another way of writing the above query is to use a regular expression with the logical OR operator (`|`):

```
$ awk '$4 ~ /^(Asia|Africa)$/ { print }' countries
```

The negation operator `!` has the highest priority, followed by `&&` and finally `||`. The `&&` and `||` operators evaluate their operands from left to right; once the condition has been found to be true or false, no more evaluation takes place.

### 8.4.5 Pattern ranges

A pattern range consists of two patterns separated by a comma, e.g.:

```
pat1, pat2{ ... }
```

In this case, the action is executed for each line between the first occurrence of *pat1* and the next occurrence of *pat2* inclusive. For example, the pattern

```
/Canada/, /Brazil/
```

matches a range of lines, starting with the first line that contains the string *Canada* up to the next line that contains the string *Brazil*:

|        |      |     |               |
|--------|------|-----|---------------|
| Canada | 3852 | 24  | North America |
| China  | 3692 | 866 | Asia          |
| USA    | 3615 | 219 | North America |
| Brazil | 3286 | 116 | South America |

Similarly, the program

```
$ awk 'FNR == 1, FNR == 5 { print FILENAME, $0 }' countries
```

outputs the first five records of each input file, prefixing the records with the name of the current input file. *FNR* is the sequence number of the current record in the current input file; *FILENAME* is the name of the current input file. Table 98 summarizes the special variables.

## 8.5 Actions

The action in a pattern-action statement determines how the input records selected by the pattern are to be processed. In many cases, actions are simple printing or assignment statements. However, they may also be more complex actions consisting of two or more statements. This section first shows you how to program simple actions using special and user-defined variables. You will then learn how to program more complex actions.

## 8.5.1 Processing the input file with simple actions

The simplest action that an *awk* program can execute is to print each input line using the *print* statement. As you can see, an *awk* program may consist of a single action. You must always enclose the action itself in braces *{}*.

```
$ awk '{ print }' countries
```

The *print* action is the same as the *print \$0* action. It prints each record in *countries* file.

### Special variables

As well as reading in the input lines and dividing them into fields, *awk* determines the number of records read in and the number of fields in the current record. You can then process these values in your *awk* programs. The variable *NR* stands for the sequence number of the current record in the input; *NF* stands for the number of fields in the record. The program

```
$ awk '{ print NR, NF }' countries
```

therefore prints the sequence number of each record and the number of fields in each record, while

```
$ awk '{ print NR, $0 }' countries
```

prints each record preceded by its record number.

### User-defined variables

As well as providing special variables such as *NF* and *NR*, *awk* allows you to define your own variables for purposes such as data storage or arithmetic. Let us assume that you want to calculate the total and average population from the population data contained in the *countries* file. To do this, you create a program file called *population* with the contents:

```
{ sum = sum + $3 }
END { print "Total population is", sum, "million."
 print "Average population of" ,NR, "countries is", sum/NR,
 "million."
}
```

Then you call the *population* program, specifying *countries* as the input file:

```
$ awk -f population countries
```

The first action adds together the values of the third field to give the total population (*awk* first initializes *sum* with the numeric value *zero*). The second action, which extends over lines two and three, outputs the sum and the average. Note that the second action is not executed until the last input line of *file* has been read in, since the action is defined in the *END* pattern.

```
Total population is 2201 million.
Average population of 10 countries is 220.1 million.
```

## 8.5.2 Processing the input file with complex actions

Complex actions are actions that process the input file using special variables, arithmetic, strings and string functions, control-flow statements, arrays and user-defined functions. The following sections describe these complex actions and their applications.

### Special variables

You can access the fields of the current record using the special variables *\$1*, *\$2*, ..., *\$NF*. Special variables share all the properties of other variables. You can use them in arithmetic or string operations and assign values to them. For example, you can divide the second field of the *countries* file by *1000* to convert the area from thousands to millions of square miles:

```
$ awk '{ $2 /= 1000; print }' countries
```

You can also assign a new string to a field. To do this, you include the following statements in an *awk* program file:

```
BEGIN { FS = OFS = "\t" }
$4 == "North America" { $4 = "NA" }
$4 == "South America" { $4 = "SA" }
 { print $0 }
```

In this program, the *BEGIN* action resets the input field separator *FS* and the output field separator *OFS* to a tab. Note that the *print* function in the fourth line prints the value of *\$0* after it has been modified by previous assignments.

You can use expressions to access the fields. For example, *\$(NF-1)* accesses the second to last field of the current record. Do not omit the parentheses, since the value of *\$NF-1* is 1 less than the value of the last field.

A special variable that accesses a nonexistent field, e.g. *\$(NF+1)*, is initialized with an empty string. You can create a new field, however, simply by assigning a value to it. If, for example, you call the following program with *countries* as its input file, the program creates a fifth field giving the population density:

```
$ awk 'BEGIN { FS = OFS = "\t" }
> { $5 = 1000 * $3 / $2; print }' countries
```

The records can have different lengths, although most implementations restrict you to 100 fields per record.

Table 98 summarizes the special variables built into *awk*. You are already familiar with some of these variables; the following sections introduce you to some more of them.

| Variable | Meaning                                                                | Default       |
|----------|------------------------------------------------------------------------|---------------|
| ARGC     | Number of command-line arguments                                       | -             |
| ARGV     | Array containing the command-line arguments                            | -             |
| FILENAME | Name of the current input file                                         | -             |
| FNR      | Sequence number of the record in the current input file                | -             |
| FS       | Input field separator                                                  | Blank and tab |
| NF       | Number of fields in the current record                                 | -             |
| NR       | Sequence number of the current record in the input                     | -             |
| OFMT     | Output format for floating point numbers                               | %.6g          |
| OFS      | Output field separator                                                 | Blank         |
| ORS      | Output record separator                                                | Newline       |
| RS       | Input record separator                                                 | Newline       |
| RSTART   | Starting position of the string matched by the <i>match()</i> function | -             |
| RLENGTH  | Length of the string matched by the <i>match()</i> function            | -             |
| SUBSEP   | Subscript separator                                                    | "\ 034"       |

Table 98: awk special variables

Arithmetic

*awk* is particularly suitable for performing arithmetic. If you want to calculate numeric values, you can include standard arithmetic expressions in an action. Let us assume, for example, that you want to calculate the population density of each country in the *countries* file. Since the second field gives the area in thousands of square miles and the third field gives the population in millions, you can use the following statement to calculate the population density per square mile:

```
1000 * $3 / $2
```

The program

```
$ awk '{ printf "%10s %6.1f\n", $1, 1000 * $3 / $2 }' countries
```

when applied to the *countries* file, prints the name of each country and its population density:

```
USSR 30.3
Canada 6.2
China 234.6
USA 60.6
Brazil 35.3
Australia 4.7
India 502.0
Argentina 24.3
Sudan 19.6
Algeria 19.6
```

Arithmetic is performed internally as floating point operations. *awk* recognizes the following arithmetic operators:

| Arithmetic operators | Meaning               |
|----------------------|-----------------------|
| +                    | Addition              |
| -                    | Subtraction           |
| *                    | Multiplication        |
| /                    | Division              |
| %                    | Remaindering          |
| ^                    | Exponent (same as **) |

You can include these operators in arithmetic expressions and apply them to constants, variables, field names, array elements, functions and other expressions, all of which are described later on in this chapter. Note that *awk* recognizes and produces scientific (exponential) notation: *1e6*, *1E6*, *10e5* and *100000* all represent the same number.

*awk* has assignment statements like those found in the C programming language. The simplest form is the assignment statement:

```
v = e
```

where *v* is a variable or field name and *e* is an expression. For example, you could use the following program to calculate the number of Asian countries and their total population. Note that you can include several pattern-action statements on the same line by separating them with semicolons:



```
$ awk '$4 == "Asia" { pop = pop + $3; n = n + 1 }
> END { print "population of", n,
> "Asian countries in millions is", pop }' countries
```

If this program uses *countries* as its input file, it produces the following output:

```
population of 3 Asian countries in millions is 1765
```

The action associated with the pattern

```
$4 == Asia
```

contains two assignment statements, one to calculate the total population and the other to count the countries. Although the variables are not initialized explicitly, the program runs as intended because *awk* initializes each variable with the string value "" (empty string) and the numeric value 0.

You can use the += and ++ operators to make the assignments in the previous program more compact:

```
$4 == "Asia" { pop += $3; ++n }
```

The += operator is borrowed from the C programming language: therefore

```
pop += $3
```

is a shorter, faster way of specifying

```
pop = pop + $3
```

The same is true of the ++ operator, which increments a variable by 1.

The following table lists the abbreviated assignment operators:

| Abbreviated assignment operators | Example | Meaning             |
|----------------------------------|---------|---------------------|
| =                                | a = b   | (simple assignment) |
| +=                               | a += b  | a = a + b           |
| -=                               | a -= b  | a = a - b           |
| *=                               | a *= b  | a = a * b           |
| /=                               | a /= b  | a = a / b           |
| %=                               | a %= b  | a = a % b           |
| ^=                               | a ^= b  | a = a ^ b           |

Other operators are the increment operator

++

and the decrement operator

--

As in C, you can use both of these as prefix ( $++x$ ) or postfix ( $x++$ ) operators. If  $x$  is 1,  $i=++x$  first increments  $x$ , then assigns the value 2 to  $i$ .  $i=x++$ , on the other hand, first sets  $i$  to 1, then increments  $x$ . An analogous interpretation applies to — as a prefix or postfix operator.

You can use all assignment, increment and decrement operators in arithmetic expressions.

Default initialization is helpful in the following program, which finds the country with the highest population:

```
$ awk 'maxpop < $3 { maxpop = $3; country = $1 }
> END { print country, maxpop }' countries
```

Table 99 summarizes the built-in arithmetic functions in *awk*.

| Function       | Meaning                                             |
|----------------|-----------------------------------------------------|
| atan2( $y,x$ ) | Arctangent of $y/x$ in the range $-pi$ through $pi$ |
| cos( $x$ )     | Cosine of $x$ ( $x$ is specified in radians)        |
| exp( $x$ )     | Exponential function of $x$                         |
| int( $x$ )     | Integer part of $x$                                 |
| log( $x$ )     | Natural logarithm of $x$                            |
| rand()         | Random number between 0 and 1                       |
| sin( $x$ )     | Sine of $x$ ( $x$ is specified in radians)          |
| sqrt( $x$ )    | Square root of $x$                                  |
| srand( $x$ )   | $x$ is the seed (initial value) for rand()          |

Table 99: awk built-in arithmetic functions

$x$  and  $y$  are any expressions. The *rand()* function returns a pseudo random number (a floating point number in the range (0,1)); you can use *srand( $x$ )* to specify the seed for the random number generator. If you do not specify an argument for *srand()*, it takes its seed from the current time.

Strings and string functions

A string is a series of characters enclosed in double quotes, e.g. "abc" or "Hello everyone". A string can contain the C programming language escape sequences for control characters (see section 8.4.2 *The second type of pattern: regular expressions*).

You create strings (alphanumeric constants) by concatenating constants, variables, field names, array elements, functions and other expressions. The program

```
$ awk '{ print NR ":" $0 }' countries
```

prints each record preceded by its record number and a colon (:). There is no explicit symbol for concatenation; you simply write the operands one after the other.

Table 100 summarizes the built-in string functions in *awk*. In the table, *r* stands for a regular expression (either as a string or as */r/*), *s* and *t* stand for string expressions, and *n* and *p* stand for integers.

| Function                                          | Meaning                                                                                                                      |
|---------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------|
| gsub( <i>r</i> , <i>s</i> )                       | Substitute <i>s</i> for <i>r</i> globally in the current record, return the number of substitutions.                         |
| gsub( <i>r</i> , <i>s</i> , <i>t</i> )            | Substitute <i>s</i> for <i>r</i> globally in string <i>t</i> , return the number of substitutions.                           |
| index( <i>s</i> , <i>t</i> )                      | Return starting position of substring <i>t</i> in <i>s</i> ; return 0 if <i>s</i> does not contain the substring.            |
| length( <i>s</i> )                                | Return length of string <i>s</i> .                                                                                           |
| match( <i>s</i> , <i>r</i> )                      | Return the starting position in <i>s</i> of the string matching <i>r</i> ; return 0 if no string is found.                   |
| split( <i>s</i> , <i>a</i> )                      | Split string <i>s</i> into fields on FS and store in array <i>a</i> ; return the number of fields.                           |
| split( <i>s</i> , <i>a</i> , <i>r</i> )           | Split string <i>s</i> into fields on <i>r</i> and store in array <i>a</i> ; return the number of fields.                     |
| sprintf<br>( <i>fmt</i> ,<br><i>ausdr_liste</i> ) | Return <i>expr_list</i> formatted according to format string <i>fmt</i>                                                      |
| sub( <i>r</i> , <i>s</i> )                        | Substitute string <i>s</i> for the first occurrence of the regular expression <i>r</i> ; return the number of substitutions. |

Table 100: awk built-in string functions

| Function      | Meaning                                                                                                                           |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------|
| sub(r,s,t)    | Substitute <i>s</i> for the first occurrence of the regular expression <i>r</i> in <i>t</i> ; return the number of substitutions. |
| substr(s,p)   | Return substring of <i>s</i> from position <i>p</i> to end.                                                                       |
| substr(s,p,n) | Return substring of <i>s</i> starting at position <i>p</i> with length <i>n</i> .                                                 |

Table 100: awk built-in string functions

The *sub* and *gsub* functions work in the same way as the substitute command in the text editor *ed* (see *ed* in the *Commands* manual [5]). The *gsub(r,s,t)* function substitutes the replacement string *s* for all substrings of the string *t* that match the regular expression *r*. As in *ed*, the first of the specified patterns is used until no more matches are found. The function returns the number of substitutions made. The *gsub(r,s)* function is the same as *gsub(r,s,\$0)*. For example, the program

```
$ awk '{ gsub(/USA/, "United States"); print }' countries
```

modifies its input by replacing each occurrence of the string *USA* with the string *United States*. The *sub* function is similar, but replaces only the first occurrence of the matching substring.

The *index(s,t)* function returns the position of the first character of substring *t* in *s*. If *t* does not occur in *s*, the function returns the value 0. The first character of a string is said to be at position 1. The function

```
index("banana", "an")
```

therefore returns the value 2.

The *length()* function returns the number of characters in the string you specify as the argument. For example,

```
{ print length($0), $0 }
```

prints each record preceded by its length, including the blanks between the input fields of a record but not the input record separator. If you specify *countries* as the input file, the program

```
$ awk 'length($1) > max { max = length($1); name = $1 }
> END { print name }' countries
```

prints the longest country name: *Australia*.

The *match(s,r)* function returns the starting position in string *s* where the regular expression *r* occurs, or 0 if it does not occur. This function also sets the two special variables *RSTART* and *RLENGTH*. *RSTART* is set to the starting position

of the match in the string, i.e. the value returned by the function. *RLENGTH* is set to the length of the matched substring. If there is no match, *RSTART* is set to 0 and *RLENGTH* to -1. For example, the following program finds the first occurrence of the lowercase letter *i*, followed by a maximum of one character, followed by the lowercase letter *a* (see the section on *Regular expressions* in chapter 13 Appendix):

```
$ awk '{ if (match($0, /i.?a/))
> print RSTART, RLENGTH, $0
> }' countries
```

The program produces the following output:

|    |   |           |      |      |     |               |
|----|---|-----------|------|------|-----|---------------|
| 17 | 2 | USSR      | 8650 | 262  |     | Asia          |
| 26 | 3 | Canada    |      | 3852 | 24  | North America |
| 3  | 3 | China     |      | 3692 | 866 | Asia          |
| 24 | 3 | USA       |      | 3615 | 219 | North America |
| 27 | 3 | Brazil    |      | 3286 | 116 | South America |
| 8  | 2 | Australia |      | 2968 | 14  | Australia     |
| 4  | 2 | India     |      | 1269 | 637 | Asia          |
| 7  | 3 | Argentina |      | 1072 | 26  | South America |
| 17 | 3 | Sudan     |      | 968  | 19  | Africa        |
| 6  | 2 | Algeria   |      | 920  | 18  | Africa        |

*match()* matches the leftmost longest matching string. For example, with the record

```
AsiaaaaAsiaaaaaan
```

as input, the program

```
{ if (match($0, /a+/)) print RSTART, RLENGTH, $0 }
```

finds the first occurrence of the letter *a* in the input record, then sets *RSTART* to 4 and *RLENGTH* to 3. *a+* means that the letter *a* may occur once or more than once.

The *sprintf(format, expr1, expr2,..., exprn)* function formats a string consisting of the expressions *expr1, expr2,..., exprn* according to the specifications in the *format* string. This function does not, however, output the string itself. The description of the *printf* function in section 8.7 *Output* includes a summary of the format specifications supported. The statement

```
x = sprintf("%10s %6d", $1, $2)
```

assigns to *x* the string produced by formatting the values of *\$1* and *\$2* as a ten-digit string and a decimal number in a field with a minimum width of six characters. You can then use *x* in any subsequent computation.

The *substr(s,p,n)* function returns the substrings in *s* that start at position *p* and have a maximum length of *n* characters. If you use the *substr(s,p)* variation, the substring must end at the same position as *s*; in other words, the substring is the suffix of *s* beginning at position *p*. For example, you can abbreviate the country names in the input file *countries* to their first three characters using the program:

```
$ awk '{ $1 = substr($1, 1, 3); print }' countries
```

This program produces the following output:

```
USS 8650 262 Asia
Can 3852 24 North America
Chi 3692 866 Asia
USA 3615 219 North America
Bra 3286 116 South America
Aus 2968 14 Australia
Ind 1269 637 Asia
Arg 1072 26 South America
Sud 968 19 Africa
Alg 920 18 Africa
```

Note that setting *\$1* forces *awk* to recompute the entire record, i.e. *\$0*. The fields are therefore separated by blanks (the default value of the special variable *OFS*) and not by tabs.

If you want the strings to be concatenated on output, simply write them one after the other without an operator in the expression. For example, if you call the program

```
$ awk '{ s = s substr($1, 1, 3) " " } END { print s }' countries
```

with *countries* as its input, you will obtain the following output:

```
Can Chi USA Bra Aus Ind Arg Sud Alg
```

*s* is built up one element at a time from an initially empty string.

## Number or string

Variables, fields and expressions can have a numeric or alphanumeric (string) value. They take on numeric or string values according to context. For example, in the context of an arithmetic expression like

```
pop += $3
```

*pop* and *\$3* must be treated numerically; if necessary, their values will be converted to numeric type.

In the context of a string function like

```
print $1 ":" $2
```

*\$1* and *\$2* must be strings in order to be concatenated. In this case, too, the data type is converted if necessary.

In an assignment with the format  $v = a$  or  $v\ op=a$ ,  $v$  assumes the same data type as  $a$ . In an ambiguous context such as

```
$1 == $2
```

the type of the comparison depends on whether the fields are numeric or string. This cannot be determined until the program actually executes, since the data type may differ from record to record.

If both operands are numeric, a numeric comparison is made. Otherwise, the operands are converted to strings to enable a string comparison to take place. Special variables are always strings. In addition, each field that contains only a number is considered to be numeric. The data type is determined during program execution. For example, the comparison  $\$1 == \$2$  will succeed on any pair of the inputs

```
11.0+10.1e+110E-1001
```

but will fail on the following inputs:

```
(null) 0
(null) 0.0
0a 0
1e50 1.0e50
```

There are two ways of converting an expression to a different data type:

| Format            | Meaning                                                                                 |
|-------------------|-----------------------------------------------------------------------------------------|
| <i>number</i> ""  | Concatenate a <i>number</i> with an empty string to convert it to the string data type. |
| <i>string</i> + 0 | adds <i>zero</i> to a <i>string</i> to convert it to the numeric data type.             |

If you want to force a string comparison between two fields, you enter a statement in the format:

```
$1 "" == $2 ""
```

The numeric value of a string is the value of any prefix of the string that looks numeric. For example, the numeric value of

```
12.34x
```

is *12.34*, while the numeric value of

```
x12.34
```

is zero. You determine the string value of an arithmetic expression by formatting the string with the output format conversion *OFMT*.

Uninitialized variables are assigned the numeric value 0 or the string value "" (empty string). Nonexistent fields and fields that are explicitly empty have the string value "" only; they are not numeric.

## Control-flow statements

*awk* provides you with the control-flow statements *if-else*, *while*, *do-while* and *for*, and allows you to group statements together using braces as in the C programming language.

The *if* statement has the following format:

```
if (expression) statement1 else statement2
```

An expression can contain the relational operators <, <=, >, >=, ==, and !=, the regular expression matching operators ~ and !~, and the logical operators ||, && and !. In addition, you can concatenate expressions by juxtaposing them and use parentheses to group them together.

In an *if* statement, *awk* first evaluates the *expression*. If *expression* is not zero, *statement1* is executed; otherwise *statement2* is executed. The *else* part is optional.

Instead of a single statement, you can specify a list of statements enclosed in braces. If you do so, you must separate the individual statements from one another using newlines or semicolons.

You can rewrite the program for calculating the highest population, described in the subsection on *Arithmetic*, using an *if* statement as follows:

```
$ awk '{ if (maxpop < $3) {
> maxpop = $3
> country = $1
> }
> }
> END { print country, maxpop }' countries
```

Unlike the *if-else* statement, the following control-flow statements are loop constructs. Whereas higher programming languages use loop constructs mostly for file processing, e.g. to read to the end of a file, *awk* always reads all the records in an input file, even if the program does not contain a loop construct. You use these loop constructs only if you want to go to another part of the program after reading a record in the input file.



The *while* statement corresponds to the C statement with the same name:

*while (expression) statement*

The *statement* is repeated and the *expression* evaluated as long as *expression* is neither zero nor an empty string. For example, the following program repeats the *print* action as long as *i* is less than or equal to the *NF* variable. The condition specified in *expression* is not satisfied until the value of *i* is greater than *NF*. Once the condition is satisfied, no further iteration takes place. In the following example, all the input fields of the *countries* file are output on separate lines:

```
$ awk ' { i = 1
> while (i <= NF)
> { print $i i++ }
> }' countries
```

The *for* statement corresponds to the C statement with the same name:

*for (expression1; expression; expression2) statement*

It has the same function as

*expression1*

*while (expression) {*

*statement*

*expression2*

*}*

The program

```
$ awk '{ for (i = 1; i <= NF; i++) print $i }' countries
```

therefore produces the same results as the *while* example shown above. An alternate version of the *for* statement is described in the next section.

The *do-while* statement has the following format:

*do statement while (expression)*

The *statement* is repeated until the value of the *expression* is 0. Since the test takes place after *statement* is executed (at the end of the loop), it is always executed at least once. As a result, the *do* statement is used much less often than *while* or *for*, which test for completion at the start of the loop.

The following *do* statement in an *awk* program file prints all lines except those between *start* and *stop*:

```
/start/ {
 do {
 getline x
 } while (x !~ /stop/)
}
{ print }
```

The *break* statement causes an immediate exit from an enclosing *while* or *for* statement; the *continue* statement causes the next iteration to begin. The *next* statement causes *awk* to skip immediately to the next record and begin matching patterns starting from the first pattern-action statement.

The *exit* statement causes the program to behave as if it had reached the end of the input. No more input is read and the *END* action, if specified, is executed. If the *END* action contains the statement

```
exit expr
```

the program returns the value of *expr* as its exit status. If there is no *expr*, the exit status is 0.

## One-dimensional arrays

*awk* provides one-dimensional arrays. You do not have to declare arrays and array elements; like variables, they come into existence simply by being mentioned. An array subscript can be a number or a string. The following example contains a conventional numeric subscript; the statement

```
x[NR] = $0
```

assigns the current input line to the *NR*th element of the array *x*. In fact, it is possible in principle (although somewhat slow) to read the entire input into an array using the following *awk* program:

```
$ awk '{ x[NR] = $0 }
> END { ##OK . . . processing . . . }' file
```

The first action merely records each input line in the array *x*, indexed by line number. Processing takes place in the *END* statement.

You can also use non-numeric values to reference array elements. For example, the statements in the following program file accumulate the total population of Asia and Africa into the array *pop*. The *END* action then prints the total population of these two continents:

```
$ awk '/Asia/ { pop["Asia"] += $3 }
 /Africa/ { pop["Africa"] += $3 }
 END { print "Asian population in millions is", pop["Asia"]
> print "African population in millions is", pop["Africa"]
> }' countries
```

The program produces the following output:

```
Asian population in millions is 1765
African population in millions is 37
```

If, in this program, you had used *pop[Asia]* instead of *pop["Asia"]*, the expression would have used the value of the variable *Asia* as the array subscript. Since the variable is uninitialized, these values would have been accumulated in *pop[""]*.

Let us assume that you want to determine the total area of each continent in the input file *countries*. You can use any expression as the subscript provided the subscript is a field in the input file. For example, the statement

```
area[$4] += $2
```

uses the string in the fourth field of the current input record to index the *area* array and accumulates the value of the second field in that entry:

```
$ awk 'BEGIN { FS = "\t" }
> { area[$4] += $2 }
> END { for (name in area)
> print name, area[name] }' countries
```

If you use *countries* as the input file, this program produces the following output:

```
Africa 1888
North America 7467
South America 4358
Asia 13611
Australia 2968
```

This program uses a variation of the *for* statement that iterates over all defined subscripts of an array:

*for (i in array) statement*

executes *statement*, setting the variable *i* in turn to each value of *i* for which *array[i]* has been defined. The loop is executed once for each defined subscript, selected in a random order. Results are unpredictable if *i* or *array* is altered during the loop.

*awk* does not support multidimensional arrays, but does permit a list of subscripts. They are combined into a single subscript, with the values separated by an unusual string that is stored in the *SUBSEP* variable. For example, the statements

```
for (i = 1; i <= 10; i++)
 for (j = 1; j <= 10; j++)
 arr[i,j] = ...
```

create an array that behaves like a two-dimensional array; the subscript is formed by concatenating *i*, *SUBSEP* and *j*.

You can check whether a particular subscript *i* occurs in an array *arr* by testing the condition *i* in *arr*, as in the following example:

```
if ("Africa" in area) ...
```

This statement tests the condition without creating *area["Africa"]*, which would happen if you used the following statement:

```
if (area["Africa"] != "") ...
```

Note that neither of these statements tests whether the array *area* contains an element with the value *"Africa"*.

The built-in function *split* allows you to split any string into fields in the elements of an array. The function

```
split("s1:s2:s3", a, ":")
```

splits the string *s1:s2:s3* into three fields, separated by the field separator *:*, and stores *s1* in *a[1]*, *s2* in *a[2]* and *s3* in *a[3]*. The number of fields found is returned as the value of the *split* function. The third argument of *split* is a regular expression to be used as the field separator. If you do not specify a third argument, *FS* is used as the field separator.

You can delete an array element using the *delete* statement:

```
delete array_name[subscript]
```

## User-defined functions

*awk* enables users to define their own functions. A function is defined as:

```
function name (argument_list) { statements }
```

You can define a function anywhere a pattern-action statement can occur. The argument list consists of a series of variable names separated by commas. If you specify these variables within the body of the function, they refer to the actual parameters when the function is called. Do not enter a blank between the function name and the left parenthesis of the argument list when you call the

function, otherwise *awk* will interpret it as a concatenation. For example, the statements in the following program file define and test the recursive factorial function:

```
function fact(n) {
 if (n <= 1)
 return 1
 else
 return n * fact(n-1)
 }
 { print $1 "! is " fact($1) }
```

You pass array arguments by reference, as in C. This makes it possible for the function to alter array elements or create new ones. Unstructured (scalar) arguments are passed as values, however; the values outside the function therefore remain unchanged. The parameters within a function are local variables, but all other parameters are global. You can have any number of extra formal parameters that are used only as local variables. You may omit the *return* statement but, if you do, the returned value is undefined.

### awk program format

An *awk* program may also contain comments. A comment begins with the character *#* and ends at the end of a line:

```
print x,y# comment
```

A statement in an *awk* program normally occupies a single line. If a line contains several statements, they must be separated by semicolons. It is advisable to divide a long statement over two or more lines, ending each line with a backslash (*\*). A string enclosed in double quotes ("*...*") cannot be divided and continued on the next line. It is not normally necessary to continue a line explicitly in this way, however, since a statement is continued automatically if the line ends in a comma. It may be necessary, for example, in a *print* or *printf* statement or after the *&&* and *||* operators.

# 8.6 Input

You usually pass input data to an *awk* program by specifying the file name(s) when you call the program. This is the method used in this chapter, but there are also several other methods. For example, you can pipe the output of a program to *awk*. This section describes the various ways in which you can pass input to an *awk* program.

*awk* processes the input lines or records sequentially. Records are character strings delimited by newlines. You enter a newline by pressing `↵`. *awk* then divides each record into fields. A field is normally a string that does not contain any blanks or tabs.

The input file used here is *countries* with which you are already familiar. The contents of this file are listed below to refresh your memory:

|           |      |     |               |
|-----------|------|-----|---------------|
| USSR      | 8650 | 262 | Asia          |
| Canada    | 3852 | 24  | North America |
| China     | 3692 | 866 | Asia          |
| USA       | 3615 | 219 | North America |
| Brazil    | 3286 | 116 | South America |
| Australia | 2968 | 14  | Australia     |
| India     | 1269 | 637 | Asia          |
| Argentina | 1072 | 26  | South America |
| Sudan     | 968  | 19  | Africa        |
| Algeria   | 920  | 18  | Africa        |

The number of fields in a record depends on the number of field separators. By default, the field separator is any number of blanks or tabs. For example, the first record in the *countries* file consists of four fields, the second record consists of five fields, the third record consists of four fields, etc. You can also use the tab on its own as the field separator so that each line contains four fields. This option is described later on. The first field of a line is assigned the variable `$1`, the second field is assigned the variable `$2` and so on. The variable `$0` is assigned to the record as a whole.

## 8.6.1 Files and pipes

You can pass data to an *awk* program by writing it to a file and then instructing the program to process the file. In the following example, *awk* takes data from a file called *input\_file*:

```
$ awk '{print $0}' input_file
```

Even if you are working with a program file, you must still specify the input file as usual when calling the program. It is always the last parameter on the command line:

```
$ awk -f prog1 input_file
```

If you do not specify a file name, *awk* reads from standard input (see section 8.3 *Calling awk*).

Another common option is to have the output of a second program piped to *awk*. For example, you can use the *egrep* command to find lines that contain a particular regular expression. The pipe

```
egrep 'Asia' countries | awk '...'
```

uses *egrep* to select the lines that contain the string *Asia* and then pass these lines to the *awk* program for subsequent processing.

## 8.6.2 Input separators

According to the default setting of *FS*, the input field separator, input fields are separated by blanks or tabs. Since leading blanks are suppressed, the following lines all have the same first field:

```
field1 field2
field1
field1
```

You can set the field separator to any regular expression by assigning a value to the *FS* special variable. For example, you can use the following statement in an *awk* program file to set the field separator to *&*:

```
BEGIN { FS = "&" }
```

This overrides the default field separator, replacing it with *&*.

You can, of course, assign more than one value to *FS*. The values you specify override the default field separators. For example, the statement

```
BEGIN { FS = ",[\\t]*|([\\t]+)" }
```

specifies that the field separator is any string consisting of a comma followed by a blank or tab, or any string consisting of blanks or tabs with no comma.

You can also set *FS* on the command line using the *-F* argument. Similarly, you can assign several values to the field separator. For example, the following command assigns the characters *+* and *\$* as field separators:

```
$ awk -F'(+[$])' -f prog text
```

**Note:**

If you assign the blank to *FS* explicitly, e.g. by specifying *FS=" "*, both blanks and tabs are interpreted as field separators. If, however, you assign the tab explicitly (*FS=\t*), only tabs, not blanks, are interpreted as field separators.

### 8.6.3 Multiline records

Records are normally separated by newlines (*\n*), making each line a record. You can change this by assigning an empty string to *RS*, the special variable for the record separator:

```
BEGIN{ RS = " " }
```

The input records can now be several lines long; the records are separated by blank lines. A common way of processing multiline records is to use the statement

```
BEGIN{ RS = ""; FS = "\n" }
```

This statement sets the record separator to a blank line and the field separator to a newline. Each line is then one field. However, the maximum length of a record is 2500 characters. The subsection on *The getline function* and section 8.8 *Using awk with other commands and the shell* contain more examples of processing multiline records.

### 8.6.4 The getline function

*awk*'s facility for automatically dividing its input into records that are more than one line long is not adequate for some tasks. For example, if records are not separated by blank lines but by a more complicated record separator, it is not enough simply to set *RS* to an empty string. In such cases, the program must break each record into fields. Here are some suggestions.

You can use the *getline* function to read input from the current input, a file or a pipe. In the latter two cases, you redirect input as for *printf*.

The *getline* function on its own reads the next, i.e. every other, input record into *\$0* and splits it into fields. At the same time, it resets the variables *NF*, *NR* and *FNR*. To see how *getline* works, create a program file *p0* with the following contents:

```
{ getline
 print "NR= " NR, "FNR= " FNR, "NF= " NF
 print $1 }
```



Now call the *p0* program with *countries* as the input file and see what is output:

```
$ awk -f p0 countries
NR=2 FNR=2 NF=6
Canada
NR=4 FNR=4 NF=6
USA
NR=6 FNR=6 NF=5
Australia
NR=8 FNR=8 NF=6
Argentina
NR=10 FNR=10 NF=5
Algeria
$
```

### The statement

```
getline x
```

also reads the next record (i.e. every other record) into the variable *x* and resets *NF* and *FNR*. The statement

```
getline <"file1"
```

reads a record from *file1* into *\$0*. In addition to *file1* which is read by *getline*, you must still specify an input file when you call the program. In the following example, the input file is called *file2*. Call the program:

```
$ awk '{getline <"file1"
 print $0, NF, FNR, NR }' file2
```

For each input record from *file2*, *getline* reads a record from *file1* into *\$0*. If *file2* contains ten input records and *file1* only eight, the program first outputs eight records from *file1*. It then takes the last two records to be output from *file2*. In addition, it resets the variables *NF*, *FNR* and *NR*. The statement

```
getline x <"file1"
```

reads a record from *file1* into the variable *x*. It does not reset the variables *NR*, *NF* and *FNR*.

If you specify an expression as a file name, you should enclose it in parentheses for evaluation, since *<* takes precedence over concatenation:

```
while (getline x < (ARGV[1] ARGV[2])) { ... }
```

Without parentheses, a statement such as

```
getline x < "tmp" FILENAME
```

sets the variable *x* to read the *tmp* file instead of *tmp* <*\$FILENAME*>. Another disadvantage of this variation of the *getline* statement is that a statement like

```
while (getline x < file) { ... }
```

creates an endless loop if the file cannot be read in since *getline* returns *-1* in the event of an error. *awk* recognizes the following return values:

| Return value | Meaning                            |
|--------------|------------------------------------|
| 1            | No errors occurred during reading. |
| 0            | End of file.                       |
| -1           | Error (e.g. record not found).     |

You can check the return value of a file that has been read in as follows:

```
return_value = getline < "file1"
if return_value <= 0 ...
```

You can also pipe the output of another command directly to *getline*. For example, the program

```
$ awk '{ while ("who" | getline)
 n++;
 END { print n }' file
```

executes the *who* command and pipes its output to *getline*. Each iteration of the *while* loop reads in one more line and increments the variable *n* by *1*; after termination of the *while* loop, *n* contains the number of users. Note that the program is executed for each input record in the input file *file*. Similarly, the program

```
$ awk '{ "date" | getline d
 > print d }' file
```

pipes the output of the *date* command to the variable *d*, thus setting *d* to the current date. Note that the date is displayed in field *d* for each input record from *file*.

Table 101 summarizes the *getline* function.

| Form                                | Sets the variables                              |
|-------------------------------------|-------------------------------------------------|
| <i>getline</i>                      | <i>\$0</i> , <i>NR</i> , <i>FNR</i> , <i>NF</i> |
| <i>getline var</i>                  | <i>var</i> , <i>NR</i> , <i>FNR</i>             |
| <i>getline &lt;file</i>             | <i>\$0</i> , <i>NF</i>                          |
| <i>getline var &lt;file</i>         | <i>var</i>                                      |
| <i>command</i>   <i>getline</i>     | <i>\$0</i> , <i>NF</i>                          |
| <i>command</i>   <i>getline var</i> | <i>var</i>                                      |

Table 101: The awk function *getline*

### 8.6.5 Command-line arguments

The command-line arguments are passed to an *awk* program as follows: the array *ARGV* contains the elements *ARGV[0]*, . . . *ARGV[ARGC-1]* of the command line. As in the C programming language, *ARGC* contains the number of elements in the *ARGV* array. *ARGV[0]* is the name of the program, i.e. *awk*; the remaining arguments are whatever was specified on the command line when *awk* was called (excluding the program file and the options). The following command line calls an *awk* program that prints the program name followed by the arguments:

```
$ awk '
> BEGIN {
> for (i = 0; i < ARGC; i++)
> printf "%s ", ARGV[i]
> printf "\n"
> }' file1 file2
```

The program prints *awk* for *ARGV[0]*, *file1* for *ARGV[1]* and *file2* for *ARGV[2]*. Because *ARGC* contains the number 3 for the arguments *awk*, *file1* and *file2*, the count *i* assumes the values 0, 1 and 2.

One exception to the rule that an argument is a file name is when it is in the form

```
var=value
```

In this case, the variable *var* is set to *value*, as if by assignment. Such an argument is not treated as a file name. If *value* is a string, you need not use double quotes (").

## 8.7 Output

You can use the *print* and *printf* statements to generate simple output. *awk* writes to standard output unless you redirect. Like the shell, *awk* also allows you to redirect output, so that data from the *print* and *printf* functions can be directed to files and pipes. You use the *print* statement to generate simple output with the minimum of formatting; the *printf* statement produces a more carefully formatted output. This section shows you how to use these two functions.

### 8.7.1 Simple output using the print function

The easiest way to output something in the *awk* programming language is to use the *print* function. The following program consists simply of a *print* statement. The *print* function is an abbreviation for *print \$0*. You must enclose it in braces:

```
{ print }
```

The command line

```
$ awk '{ print }' countries
```

outputs each line of the *countries* file by copying the file to standard output. You can also use the *print* function to output parts of a record. For example, you could include the statement

```
{ print $1, $3 }
```

in an *awk* program to output the first and third field of each record. The *awk* program

```
$ awk '{ print $1, $3 }' countries
```

produces the following output

```
USSR 262
Canada 24
China 866
USA 219
Brazil 116
Australia 14
India 637
Argentina 26
Sudan 19
Algeria 18
```

The elements that were separated by commas in the *print* statement are separated in the output by the output field separator (default: a single blank). Each line of output ends with the output record separator (default: a single newline `\n`).

The function

```
print expr1, expr2, ..., exprn
```

prints the string value of all expressions, separated by the output field separator.

To print an empty line, you use the statement

```
{ print "" }
```

If you do not want to print every line in a file, you specify a pattern in the *print* or *printf* statement:

```
$ awk '$4 ~ /Asia/ { print $1 }' countries
```

This program prints the first field of each line in which the fourth field contains the string *Asia*. See section 8.4 *Patterns*.

8.7.2 Formatted output using the printf function

The *printf* function provides you with more options for formatting your output than the *print* function. The *printf* function is the same in *awk* as in C, except that it does not support the asterisk (\*) as a format specifier. The *printf* statement has the following general format:

```
printf format, expr1 expr2, ..., exprn
```

This statement formats *expr1, expr2, ..., exprn* according to the format specifier *format*. *format* is a string that contains both information to be printed and specifications on what conversions are to be performed on the expressions in the argument list. Each format specification begins with % and ends with a letter that stands for the type of conversion. The format specifications can be:

| Format    | Meaning                                                                                                                   |
|-----------|---------------------------------------------------------------------------------------------------------------------------|
| -         | Left-justify expression in field.                                                                                         |
| <i>fb</i> | Minimum field width. Pad field to this width if necessary. If <i>fb</i> is preceded by 0, the field is padded with zeros. |
| <i>.g</i> | Maximum string length or number of decimal places.                                                                        |

For example, the *awk* statement

```
{ printf "%10s %6d\n", $1, $3 }
```

prints the first field (*\$1*) as a right-justified string of ten characters, followed by a blank, the third field (*\$3*) as a decimal number in a six-digit field and a newline. If you now specify *countries* as the input file for this program and enter the command line shown below:

```
$ awk '{ printf "%10s %6d\n", $1, $3 }' countries
```

the program creates a formatted table:

|           |     |
|-----------|-----|
| USSR      | 262 |
| Canada    | 24  |
| China     | 866 |
| USA       | 219 |
| Brazil    | 116 |
| Australia | 14  |
| India     | 637 |
| Argentina | 26  |
| Sudan     | 19  |
| Algeria   | 18  |

The *printf* function does not generate the output record separators or newlines automatically. You must therefore include them explicitly in the format specification, e.g. as *\n*.

Table 102 summarizes the *printf* formatting characters.

| Character | Prints the expression as:                                                                                         |
|-----------|-------------------------------------------------------------------------------------------------------------------|
| c         | a single character                                                                                                |
| d         | a decimal number                                                                                                  |
| e         | <code>[-]d.dddddE[+-]dd</code>                                                                                    |
| f         | <code>[-]ddd.ddddd</code>                                                                                         |
| g         | a floating point number in <i>e</i> or <i>f</i> format, whichever is shorter, and suppresses nonsignificant zeros |
| o         | an unsigned octal number                                                                                          |
| s         | a string                                                                                                          |
| x         | an unsigned hexadecimal number                                                                                    |
| %         | prints % without converting any arguments                                                                         |

Table 102: awk printf conversion characters

Below are some examples of *printf* statements and their output:

```
printf "%d", 99/2 49
printf "%e", 99/2 4.950000e+01
printf "%f", 99/2 49.500000
printf "%6.2f", 99/2 49.50
printf "%g", 99/2 49.5
printf "%o", 99 143
printf "%06o", 99 000143
printf "%x", 99 63
printf "%s", "January" |January|
printf "%10s", "January" |January|
printf "%-10s", "January" |January|
printf "%.3s", "January" |Jan|
printf "%10.3s", "January" |Jan|
printf "%-10.3s", "January"|Jan|
printf "%%" %
```

The default output format for numbers is `%.6g`. You can change this default by assigning a new value to the *OFMT* variable. *OFMT* also controls the conversion of numeric values to strings for concatenation and the creation of array subscripts.

## 8.7.3 Output separators

The output field and record separators are stored in the special variables *OFS* and *ORS*. The default setting for *OFS* is a single blank, while the default setting for *ORS* is a single newline; however, you can reset these variables at any time. For example, the following program prints the first and second field of each record, using the colon (:) as the field separator and two newlines at the end of the second field:

```
$ awk 'BEGIN { OFS = ":"; ORS = "\n\n" }
> { print $1, $2 }' input_file
```

If the fields you want to output are not separated by commas in the *print* statement, they are printed without field separators:

```
{ print $1 $2 }
```

In the above statement, therefore, *\$1* and *\$2* are interpreted as a string produced by concatenating the fields *\$1* and *\$2*.

## 8.7.4 Redirecting output to files

You can write output to files instead of to standard output by using the redirection operators (`>`) and (`>>`). If, for example, you use *countries* as the input file for the following program, all the lines in which the third field contains a value greater than *100* are written to the *bigpop* file. The remaining lines are redirected to the *smallpop* file:

```
$ awk '$3 > 100 { print $1, $3 >"bigpop" }
> $3 <= 100 { print $1, $3 >"smallpop" }' countries
```

Note that you must enclose the file names in double quotes ("`"`) otherwise *bigpop* and *smallpop* are treated as uninitialized variables. If the names of the output files are the result of an expression, they must also be enclosed in parentheses, since the redirection operator (`>`) takes precedence over concatenation:

```
$ awk '$4 ~ /Asia/ { print $1 > ("tmp" FILENAME) }' countries

$ cat tmpcountries
USSR
China
India
$
```

The *FILENAME* variable always contains the name of the current input file, in this case *countries*. This means that the name of the output file in the above example would be *tmpcountries*.

### Caution:

Files are opened only once in an *awk* program. If you use the `>` operator, the original file contents are overwritten. If you use the `>>` operator, the original file contents remain intact with the program output appended to the file.

## 8.7.5 Piping output to other commands

You can also direct the output of the *print* command to another command to the right of the pipe symbol. The statement

```
print | "command_line"
```

pipes the output of the *print* function to *command\_line*.

The *command\_line* and file names, although shown here as literal strings enclosed in double quotes ("`"`), can also be variables or the return values of functions.



Let us assume that you want a list of continent-population pairs, sorted according to the first letter of the continent. The following *awk* program accumulates the population values in the third field for each of the continent names in the fourth field in an array called *pop*. Then it prints each continent and its population, and pipes this output into the *sort* command. To do this, you create a program file called *population* containing the following statements:

```
BEGIN { FS = "\t" }
 { pop[$4] += $3 }
END { for (c in pop)
 print c ":" pop[c] | "sort"
 }
```

Then call this program file with *awk*:

```
$ awk -f population countries
```

With *countries* as its input file, this program produces the following output:

```
Africa:37
Asia:1765
Australia:14
North America:243
South America:142
```

In all *print* statements involving redirection of output, the files or pipes are identified by their names (i.e. the above pipe has the name *sort*), but they are created and opened only once during the entire program run. In the last example, therefore, only one *sort* pipe was opened for all *c* in *pop*.

Only a limited number of files can be open at the same time in an *awk* program, as explained under *Restrictions* in section 8.10 *awk summary*. The statement

```
close ("file")
```

closes a file or pipe. *file* is the string that was originally used to open the file. When you open or close a file, different strings are treated as different commands.

## 8.8 Using awk with other commands and the shell

*awk* is most powerful when you use it in conjunction with other programs. This section describes some of the ways in which *awk* programs interoperate with other programs.

### 8.8.1 The system function

The built-in *system(command\_line)* function executes the *command\_line* command, which may be a string formatted by the built-in *sprint* function, for example. The value returned by *system* is the return status of the command issued. The *system* function allows you to use Reliant UNIX commands within your *awk* program. For example, the following program calls the Reliant UNIX *ls* command and executes it once:

```
$ awk 'BEGIN {system("ls"); print}' file
a-file
b-file
c-file
$
```

### 8.8.2 awk and the shell

In most of the examples so far, the *awk* program was in a file. You either loaded this file using the *-f* option or entered it on the command line enclosed in single quotes (*'*), as shown below:

```
$ awk '{ print $1 }' . . .
```

Because *awk* uses many of the same characters with special meaning as the shell (e.g. *\$* and *"*), you must enclose the *awk* program in single quotes (*'*) to ensure that the shell will pass the entire program unchanged to the *awk* interpreter.

Let us assume that you want a command called *addr* to search an *address\_list* for names, addresses and telephone numbers. The entries in *address\_list* are records that consist of more than one line. A typical record is:

```
G. R. Emlin
600 Mountain Avenue
Murray Hill, NJ 07974
201-555-1234
```

Records are separated by a single blank line. You want to search through the address list using commands like

```
addr Emlin
```

You can do this easily using a program such as

```
$ awk '
> BEGIN { RS = "" }
> /Emlin/
> ' address_list
```

However, you want the program to use a different search pattern each time it is executed. There are several ways of solving this problem. Create a file called *addr* with the following contents:

```
awk '
 BEGIN { RS = "" }
 /'$1'/
 ' address_list
```

The single quotes (') that enclose *\$1* are critical here. The *awk* program is only one argument, even though there are two sets of single quotes. This is because the single quotes enclosing the *\$1* variable are part of the regular expression /'\$1'/ . This enables the shell to evaluate *\$1*, in this case replacing the variable with the pattern *Emlin* as soon as you enter the command *addr Emlin*. Under Reliant UNIX, you first make *addr* executable by modifying its permissions using the command:

```
$ chmod +x addr
```

Then you call the *addr* file with the pattern *Emlin*:

```
$ addr Emlin
```

Another way of implementing the address program relies on the fact that the shell evaluates and replaces \$ parameters that are enclosed in double quotes (""):

```
$ awk "
> BEGIN { RS = \"\" }
> /$1/
> " addresslist
```

You must therefore escape the quotes (") that are assigned to the *RS* variable using backslashes (\), so that the shell passes them on to *awk* without interpreting them. However, because *\$1* is recognized as a parameter, the shell replaces it with the *pattern* when you call the *addr* program.

A third way of implementing the *addr* command is to use the *ARGV* variable to pass the regular expression to an *awk* program. This *awk* program then uses the *getline* function to search the address list explicitly:

```
$ awk 'b
> BEGIN { RS = ""
> while (getline <"addresslist")
> if ($0 ~ ARGV[1])
> print $0
> } ' $*
```

All processing takes place in the *BEGIN* statement.

Note that you can pass any regular expression to *addr*; you can retrieve information by specifying parts of an address, telephone numbers or names.

## 8.9 Sample applications

As already explained in the *Introduction*, *awk* programs are always much shorter than equivalent programs written in more conventional programming languages such as Pascal or C. This section starts by introducing you to some useful one-liners. It then presents other, more extensive, examples of *awk* programs.

### 8.9.1 Some useful one-liners

Although you can use *awk* to write large programs of some complexity, many programs are not much more complicated than the programs you have seen so far. This section presents some short statements that you might use frequently as the basis for your own programs. The statements are not explained here. If you want to test one of these statements, simply enclose it in single quotes and call *awk* together with an input file such as *countries*. Alternatively, you can include the statements in your own program file (see section 8.3 *Calling awk*).

Print the last field of each input line:

```
{ print $NF }
```

Print the tenth input line:

```
NR == 10 { print $0 }
```

Print the last input line:

```
 { line = $0 }
END { print line }
```

Print the input lines that contain more or less than four fields:

```
NF != 4 { print $0, "does not contain four fields" }
```

Print input lines that contain more than four fields:

```
NF > 4 { print }
```

Print the input lines in which the number in the last field is greater than 4:

```
$NF > 4 { print $0 }
```

Print the total number of input lines:

```
END { print NR }
```

Print the total number of fields:

```
 { nf = nf + NF }
END { print nf }
```

Print the total number of input characters:

```
 { nc = nc + length($0) }
END { print nc + NR }
```

(adding the special variable *NR* includes the number of newlines in the total).

Print the total number of lines that contain the string *Asia*:

```
/Asia/ { nlines++ }
END { print nlines }
```

(*nlines++* has the same effect as *nlines = nlines + 1* except that *nlines++* is processed faster).

## 8.9.2 Error messages

If an error occurs in an *awk* program, you receive an error message. For example, the incorrect statement (missing right parenthesis after the *\$1* variable) in the program

```
$ awk '$3 < 200 { print ($1 }' countries
```

causes the following error messages to be issued:

```
awk: syntax error at source line 1
context is
 $3 < 200 { print (>>> $1 } <<<
awk: illegal statement at source line 1
missing)
```

Some errors, such as division by zero, may be detected while your program is running. *awk* aborts the program run and outputs the sequence number of the input record involved (*NR*) and the line number in the program.

## 8.9.3 More sample applications

### Generating reports

*awk* is especially useful for generating reports that summarize and format information. For example, you can produce a report from the *countries* file that lists the countries alphabetically and lists the countries on each continent in decreasing order of population:

|                |  |     |
|----------------|--|-----|
| Africa:        |  |     |
| Sudan          |  | 19  |
| Algeria        |  | 18  |
| Asia:          |  |     |
| China          |  | 866 |
| India          |  | 637 |
| USSR           |  | 262 |
| Australia:     |  |     |
| Australia      |  | 14  |
| North America: |  |     |
| USA            |  | 219 |
| Canada         |  | 24  |
| South America: |  |     |
| Brazil         |  | 116 |
| Argentina      |  | 26  |

As with many data processing tasks, it is much easier to generate this report in several stages. First you create a list with three columns for the continent, country and population, in which each field is separated by a colon (:). To do this, you include the following *awk* statements in a file called *triples*, in which the population of a country is stored in an array called *pop*. This array is in turn indexed by subscripts in the format *continent:country*. The print statement in the *END* pattern of the program outputs the three columns, which are then piped to the *sort* program.

```
BEGIN { FS = "\t" }
 { pop[$4 ":" $1] += $3 }
END { for (cc in pop)
 print cc ":" pop[cc] | "sort -t: +0 -1 +2nr" }
```

The arguments for *sort* deserve special mention. *-t:* tells *sort* to use the colon (:) as its field separator. *+0 -1* makes the first field the primary sort key. In general, *+i -j* makes fields *i+1*, *i+2*, . . . , *j* the sort key. If *j* is omitted, the fields from *i+1* to the end of the record are used. *+2nr* makes the third field the secondary sort key (in decreasing order). *n* stands for numeric and *r* stands for reverse order. With *countries* as its input file, this program produces the following output:

```
Africa:Sudan:19
Africa:Algeria:18
Asia:China:866
Asia:India:637
Asia:USSR:262
Australia:Australia:14
North America:USA:219
North America:Canada:24
South America:Brazil:116
South America:Argentina:26
```

This output is in the right order but the wrong format. To format the output as desired, you pass it to a second *awk* program called *format*:

```

BEGIN { FS = ";" }
{
 if ($1 != prev) {
 print "\n" $1 ":"
 prev = $1
 }
 printf "\t%-10s %6d\n", $2, $3
}

```

This is a formatting program that prints only the first occurrence of a continent name and formats the lines containing the country names and population associated with this continent in the desired manner. The command line

```
$ awk -f triples countries | awk -f format
```

enables you to generate the desired report. This example illustrates how to reduce complex data transformation and formatting tasks to a few simple *awk* and *sort* commands.

## Word frequency

The first of the following examples illustrates how you use associative arrays for counting. Let us assume that you want to count the number of times each word appears in the input, where a word is any contiguous string of characters, i.e. a string containing no blanks or tabs. The following program prints the frequency of each word, sorted in descending order:

```

$ awk '{ for (w = 1; w <= NF; w++) count[$w]++ }
> END { for (w in count) print count[w],w | "sort -nr" }
> ' countries

```

The first statement uses the *count* array to accumulate the number of times each word occurs. Once the input has been read in, the second *for* loop pipes the final count along with each word into the *sort* command.

## Accumulation

Let us assume that you have two files *deposits* and *withdrawals* containing a name field and an amount field. For each name, you want to print the net balance, calculated by subtracting the withdrawals from the total deposits. The following program computes the net balance:

```

$ awk '
> FILENAME == "deposits" { balance[$1] += $2 }
> FILENAME == "withdrawals" { balance[$1] -= $2 }
> END { for (name in balance)
> print name, balance[name]
> } ' deposits withdrawals

```

The first statement uses the *balance* array to accumulate the total amount for each name in the *deposits* file. The second statement subtracts the withdrawals from each total. If only withdrawals are associated with a name, the second statement creates an entry for that name. The *END* action prints each name with its net balance.

### Random choice

The *choose* function shown below prints *k* random elements consecutively from the first *n* elements of the array *A*. *k* is the number of elements yet to be printed and *n* is the number of elements yet to be examined. The test function *rand()* < *k/n* determines whether or not the *i*th element is printed:

```
function choose(A, k, n, i) {
 for (i = 1; n > 0; i++)
 if (rand() < k/n--) {
 print A[i]
 k--
 }
}
```

### Shell-like functions

The following *awk* program is similar to the history facility of the Reliant UNIX shell. A line containing only *=* reexecutes the last command executed. A line beginning with *= command* reexecutes the last command called with the string *command*. Otherwise, the current line is executed.

```
$1 == "=" { if (NF == 1)
 system(x[NR] = x[NR-1])
 else
 for (i = NR-1; i > 0; i--)
 if (x[i] ~ $2) {
 system(x[NR] = x[i])
 break
 }
 next }
././ { system(x[NR] = $0) }
```



## Generating form letters

You can also use *awk* to generate form letters. First look at the *form.letter* file that contains the form letter:

```
This is a form letter.
The first field is $1, the second $2, the third $3.
The third is $3, second is $2, and first is $1.
```

You now want to replace the variables *\$1* through *\$3* with the following text:

```
field 1|field 2|field 3
one|two|three
a|b|c
```

The program you use to replace the variables and generate the appropriate output might look like this:

```
BEGIN { FS = "|"
 while (getline <"form.letter")
 line[++n] = $0
 }
 for (i = 1; i <= n; i++) {
 s = line[i]
 for (j = 1; j <= NF; j++)
 gsub("\\$"j, $j, s)
 print s
 }
}
```

The *BEGIN* action stores the template in the *template* array. The remaining actions process the input data, using the *gsub* function to replace template fields in the format *\$n* with the corresponding data fields.

When writing this kind of program file, you should always start with a short version and expand it, testing each new function before moving on to the next.

## 8.10 awk summary

### Command line summary

|         |                                            |
|---------|--------------------------------------------|
| awk     | <i>program file_name</i>                   |
| awk -f  | <i>program_file file_name</i>              |
| awk -Fs | Set the field separator to string <i>s</i> |
| -Ft     | Set the field separator to the tab         |

### Pattern summary

| Pattern type                        | Meaning                                                                                                   |
|-------------------------------------|-----------------------------------------------------------------------------------------------------------|
| <i>rel_expression</i>               | Compares numbers or strings using the following relational operators:<br><br><, <=, ==, !=, >=, >, ~, !~. |
| <i>/reg_expression/</i>             | Regular expression enclosed in backslashes.                                                               |
| BEGIN                               | Optional pattern type executed before the first input record is read in.                                  |
| END                                 | Optional pattern type executed after the last input record has been processed.                            |
| <i>pattern1 &amp;&amp; pattern2</i> | Compound pattern for concatenation.                                                                       |
| <i>pattern1 !! pattern2</i>         | Compound pattern for concatenation.                                                                       |
| <i>!pattern</i>                     | Negation                                                                                                  |
| <i>pattern1, pattern2</i>           | Pattern range from <i>pattern1</i> through <i>pattern2</i> .                                              |

**Summary of control-flow statements**

```

if (expr) statement [else statement]
if (subscript in array)
statement [else statement]
while (expr) statement
for (expr; expr; expr) statement
for (var in array) statement
do statement while (expr)
break
continue
next
exit [expr]

```

**Input/output summary**

|                                                    |                                                                                      |
|----------------------------------------------------|--------------------------------------------------------------------------------------|
| close ( <i>file_name</i> )                         | Close file.                                                                          |
| getline                                            | Read next input record into <i>\$0</i> ;<br>set <i>NF</i> , <i>NR</i> , <i>FNR</i> . |
| getline < <i>file</i>                              | Read next input record from <i>file</i> into <i>\$0</i> ;<br>set <i>NF</i> .         |
| getline <i>var</i>                                 | Read next input record into <i>var</i> ;<br>set <i>NR</i> , <i>FNR</i> .             |
| getline <i>var</i> < <i>file</i>                   | Read next input record from <i>file</i> into <i>var</i> .                            |
| print                                              | Print current record.                                                                |
| print <i>expr_list</i>                             | Print expressions.                                                                   |
| print <i>expr_list</i> > <i>file</i>               | Print expressions to <i>file</i> .                                                   |
| printf <i>fmt</i> , <i>expr_list</i>               | Format and print.                                                                    |
| printf <i>fmt</i> , <i>expr_list</i> > <i>file</i> | Format and print to <i>file</i> .                                                    |
| system ( <i>cmd_line</i> )                         | Execute command <i>cmd_line</i> , return status.                                     |

In the *print* and *printf* functions, *>>file* appends output to *file*, and *| command* pipes it to another command. Similarly, *command | getline* pipes output to *getline*. On reaching the end of the file, *getline* returns the value 0; in the event of an error, it returns -1.

Functions

```
func name(parameter_list) { statement }
function name(parameter_list) { statement }
function_name(expr, expr, . . .)
```

Summary of string functions

|                          |                                                                                                                                                                                                                     |
|--------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| gsub (r,s,t)             | Substitute string <i>s</i> for each substring matching regular expression <i>r</i> in string <i>t</i> ; return the number of substitutions. If <i>t</i> is omitted, the function is applied to <i>\$0</i> .         |
| index (s,t)              | Return the starting position of string <i>t</i> in string <i>s</i> ; return 0 if string is not found.                                                                                                               |
| length (s)               | Return the length of string <i>s</i> .                                                                                                                                                                              |
| match (s,r)              | Return the position of the first character of regular expression <i>r</i> in string <i>s</i> : return 0 if <i>r</i> is not found.                                                                                   |
| split (s,a,r)            | Split string <i>s</i> into fields using regular expression <i>r</i> as separator and store them in array <i>a</i> ; return the number of fields. If <i>r</i> is omitted, the field separator defined in FS is used. |
| sprintf (fmt, expr_list) | Format <i>expr_list</i> according to the format specification and print it.                                                                                                                                         |
| sub (r,s,t)              | As <i>gsub</i> , except that only the first matching substring is replaced.                                                                                                                                         |
| substr (s,i,n)           | Return the <i>n</i> -character substring of <i>s</i> that starts at position <i>i</i> ; if <i>n</i> is omitted, return the rest of the characters to the end of string <i>s</i> .                                   |

Summary of arithmetic functions

|              |                                                                                               |
|--------------|-----------------------------------------------------------------------------------------------|
| atan2 (y,x)  | Arctangent of y/x in radians.                                                                 |
| cos (expr)   | Cosine (angle in radians).                                                                    |
| exp (expr)   | Exponential.                                                                                  |
| int (expr)   | Truncate to integer.                                                                          |
| log (expr)   | Natural logarithm.                                                                            |
| rand ()      | Random number between 0 and 1.                                                                |
| sin (expr)   | Sine (angle in radians).                                                                      |
| sqrt (expr)  | Square root.                                                                                  |
| srand (expr) | New seed for random number generator; if <i>expr</i> is omitted, the seed is the time of day. |

Summary of operators (in ascending order of precedence)

|                            |                                                      |
|----------------------------|------------------------------------------------------|
| = += -= *= /= %= $\hat{=}$ | Assignment.                                          |
| ?:                         | Conditional expression.                              |
|                            | Logical OR.                                          |
| &&                         | Logical AND.                                         |
| ~ !~                       | Pattern matching operator, negation.                 |
| < <= > >= != ==            | Relational operators.                                |
| Blank                      | String concatenation.                                |
| + -                        | Addition, subtraction.                               |
| * / %                      | Multiplication, division, remaindering.              |
| + - !                      | Unary plus sign, unary minus sign, logical negation. |
| ^                          | Exponentiation (same as **).                         |
| ++ --                      | Increment, decrement (prefix and postfix).           |
| \$                         | Field.                                               |

Summary of special variables

|          |                                                                                                 |
|----------|-------------------------------------------------------------------------------------------------|
| ARGC     | Number of command-line arguments.                                                               |
| ARGV     | Array containing the command-line arguments ( <i>0..ARGC-1</i> ).                               |
| FILENAME | Name of the current input file.                                                                 |
| FNR      | Sequence number of record in current file.                                                      |
| FS       | Input field separator (default: blank).                                                         |
| NF       | Number of fields in current input record.                                                       |
| NR       | Sequence number of current record since the start of input.                                     |
| OFMT     | Output format for floating point numbers (default: <i>%.6g</i> ).                               |
| OFS      | Output field separator (default: blank).                                                        |
| ORS      | Output record separator (default: newline).                                                     |
| RS       | Input record separator (default: newline).                                                      |
| RSTART   | Index of first character matched by the <i>match()</i> function; <i>0</i> if no match is found. |
| RLENGTH  | Length of the string matched by the <i>match()</i> function; <i>-1</i> if no match is found.    |
| SUBSEP   | Separates multiple subscripts in array elements; default: <i>\ 034</i> .                        |

Restrictions

Every *awk* implementation is dependent upon certain parameters. If your *awk* program does not execute properly, you may have exceeded certain maximum parameter values. For further information refer to the description of the command *awk*.

Initialization, comparison and type conversion

Each variable and field may be a string or a number or both at any given time. If you set a variable using the assignment

*var = expr*

its type is set to that of the expression (the assignment operators are *+=*, *-=* etc.). An arithmetic expression has the numeric data type, a concatenation has the string data type, etc. If the assignment is a simple copy, as in

*v1 = v2*,

*v1* assumes the same data type as *v2*.

In comparisons, if both operands are numeric, the comparison is made numerically. Otherwise, the operands are converted to string type if necessary so that a string comparison can take place. You can convert an expression into a numeric expression by specifying

```
expr + 0
```

and you can use

```
expr ""
```

to convert an expression into a string expression (by concatenating it with an empty string).

Uninitialized variables have the numeric value 0 and the string value "". Accordingly, if *x* is uninitialized, the condition

```
if (x) ...
```

is not satisfied (is *false*), while the conditions

```
if (!x) ...
```

```
if (x == 0) ...
```

```
if (x == "") ...
```

are all true. The following condition, however, is *false*:

```
if (x == "0") ...
```

The type of a field is determined by the context wherever possible. For example,

```
$1++
```

clearly states that *\$1* is to be numeric, and

```
$1 = $1 " , " $2
```

states that both *\$1* and *\$2* are to be strings. If necessary, type conversion takes place.

In contexts where the data type is ambiguous, e.g.

```
if ($1 == $2) ...
```

the type of each field is determined on input. All fields are strings; in addition, each field that contains only a number is interpreted as numeric.

Fields that are explicitly empty have the string value ""; they are not numeric. Nonexistent fields (i.e. fields beyond *NF*) are also treated in this way.

The rules for fields also apply to fields created by the *split()* function.

You create a variable simply by specifying it in an expression. Once created, a variable is assigned the value "" as described above. For example, if *array[i]* does not currently exist, the statement

```
if (arr[i] == "") ...
```

creates it and assigns it the value "", so that the *if* condition is satisfied. The special statement

```
if (i in arr) ...
```

determines whether or not *arr[i]* exists, but does not create it if it does not already exist.



---

## 9 Network user functions

This chapter is intended for users without system administrator tasks who wish to use the functions provided by the network software of the Reliant UNIX operating system.

The various sections in this chapter are structured as follows: if a function is available through the Slfmlan user interface the section begins by explaining how to use it via Slfmlan. Then follows a description of how you can execute the same function using commands. This manual does not contain any complete command descriptions, but instead uses examples to show how you can use the communication functions. For a complete description of the network commands, see the *Network Reference Manual* [9].

The complete functional scope of the network software is only available on command level. The basic functions, however, have been implemented in the menu system.

The following is described in this chapter:

- How to use the Slfmlan user interface
- How to log in to a remote host
- How to copy files between hosts
- How to execute commands on remote hosts
- How to obtain information about remote hosts and about the network status
- Possible errors that may occur and why

### **Note:**

In the examples in this chapter, prompts from the system appear in the format *host\_name*\$ so that you can always recognize which host is currently being used.

## 9.1 Using the Sifmllan user interface

The Sifmllan interface is the component of the FACE interface for users.

Sifmllan enables users to

- communicate with remote hosts in the network
- obtain information on the network status
- copy files to or fetch them from remote hosts

You call the user interface by entering the command

`face`

You can only do this if you have been entered as a FACE user. If this is not the case, contact your system administrator.

The following menu is displayed when you call `face`:

```
1 FACE
>Office of [login]
Printer Operations
Information
Mail Services
Floppy disk
Streamer Tape
Network Operations
SINIX
Exit FACE
```

If the first time you call FACE the menu appears in English and you want to work in a different language, you can use the menu items *Office of [login]*, *Preferences* and *Set Language* to change the language used. For example, you can enter *German* or *German with umlauts* in the *Set Language* window. Any change made will not become valid until you exit and restart FACE. If you change the language, this change remains valid for every `face` call until you change the language again explicitly.

If you wish to use the network functions, select *Network Operations* from the menu shown above. The following menu is then displayed:

```
2 Network Operations
>SiLAN User
```

Select *SiLAN User*. The *LAN User Functions* menu is displayed on the screen

```
1 LAN User Functions
 Communication functions
>Login to remote computer
 Fetch file from remote computer
 Send file to remote computer
 Network Status:
 Defined computers
 Active Sessions
 Status of computers
```

The “Communication functions:” and “Network Status:” items are indented and cannot be selected - they merely subdivide the items into one group consisting of communication functions and another consisting of status information functions.

## 9.2 Logging in to a remote host

There are three ways in which you can log in to a remote host and open a session:

- by selecting the *Login to remote computer* menu item in the *Sifmllan* user interface
- by entering the `rlogin` command to log in on UNIX systems
- by entering the `telnet` command to log in on hosts running UNIX or other operating systems

### 9.2.1 Logging in to a remote host via *Sifmllan*

The *Login to remote computer* menu item enables you to open a session on a remote host. The following input form is displayed on the screen:

```
2 Connect to remote Computer Form

Name of remote computer:
Login on remote computer:
```

*Name of remote computer*

Enter the name or alias of the remote host on which you want to open a session. Press CHOICES to view a list of possible remote hosts, then press ENTER to transfer the host of your choice to the input form.

*Login on remote computer*

Enter the user (login) name. No selection list is available for this entry. The user name must be known to the remote host. If you leave this field blank, the system inserts the user name you used to access the local host.

After entering the name or alias of the remote host and your user name, press SAVE to start the login process. You will be prompted for a password unless an entry exists for you in the `$HOME/.rhosts` or `/etc/hosts.equiv` file authorizing you to work on the remote host. For information on these access permission entries, see section "Querying a password" on page 345.

If all your entries are correct, a session on the remote host you selected is opened for you under the user name you specified. You can now proceed in just the same way as when working on your local host.

## 9.2.2 Logging in to a remote host with rlogin

The `rlogin` command allows you to log in to another UNIX system in the network.

### Logging in under the same login name

If you want to work on the remote host under the same login name as the one you are using on the local host, enter `rlogin` and the name of the remote host.

If your host has been entered in the `/etc/hosts.equiv` file on the remote host, you do not need to specify a password with the `rlogin` command. There is also no need to specify a password if your host, instead of being entered in the `/etc/hosts.equiv` file on the remote host, is listed together with your login name in the `.rhosts` file of your home directory on the remote host.

If neither of these entries exists, you will be prompted to enter a password. Enter your password for the remote host and confirm it with the Return key .

If there is an entry for your system in the password database of the remote host, once you have entered the correct password you are logged in on this host just as if you had logged in at a terminal directly attached to it.

*Example:*

```
venus$ rlogin jupiter
Password: (Enter your password here.)
Last login: Mon Oct 20 00:30:52 from venus
jupiter$ pwd
/home/peter
jupiter$ exit
Connection closed.
venus$
```

**Logging in under a different login name**

You may want to log in to a remote host under a different name, for example if you want to work on another user's computer (under his or her login name), or if you are known by a different login name on the remote host. To do this you need to enter `rlogin` with the option `-l`. The format is as follows:

```
rlogin host_name -l login_name
```

If an entry exists for your host and your login in the `.rhosts` file of the home directory of the other user on the remote host, you do not need to specify a password with `rlogin`.

### 9.2.3 Terminating a session on the remote host

Normally you terminate a session with the `exit` command. If at any time you find that this command has no effect, you can try to end the session by pressing the **END** key or **CTRL** - d.

A further option for terminating the session is to enter the sequence `~.` at the beginning of a line. This terminates all your current sessions on all remote hosts, so that you are returned to your local session.

**Caution:**

Do not use `~.` to terminate a connection that has been set up via `rlogin` unless the connection cannot be terminated in the normal way at the end of the session using the `exit` command, the **END** key or **CTRL** - d. The key sequence `~.` terminates the session regardless of whether there are files open or programs running.

**Note:**

If you log in on several hosts consecutively and access each host from a different one, entering `~` terminates all connections, i.e. you will find yourself back on the local host.

*Example:*

```
venus$ rlogin comet
Last login: Thu Nov 21 05:04:03 from venus
comet% ~. [D] (Depending on the terminal emulation used,
 ~ may not be output)
Closed connection.
venus$
```

## 9.2.4 Interrupting an rlogin connection

If you are using a shell with job control (`jsh` or `ksh`), you can temporarily interrupt a connection established via `rlogin`, and resume it again later. To do this, enter a tilde (`~`) followed by the key sequence `[CTRL] - z`. The connection is interrupted and the associated `rlogin` process is stopped. You return to the system from which you logged in. To reactivate the connection, enter `fg` or `fg %job_number`. `job_number` is the job number of the interrupted process, in this case the `login` process. If you do not specify a job number, the number of the process that was last interrupted or placed in the background is used.

*Example:*

```
venus$ rlogin animation
Last login: Thu Nov 21 07:07:07 from venus
animations$ ~ [CTRL] - z (Depending on the terminal emulation used,
 ~ may not be output)

Stopped
venus$ pwd
/home/medici
venus$ fg
rlogin animation (Return gives you a prompt)
animation$ exit
Connection closed.
venus%
```

## 9.2.5 Querying a password

A user on a remote host who is not entered in the password database of a host cannot access this host.

If there is an entry for this user in the password database, this password is queried. If the password entered matches the one in the database, the user can access the host.

In the following cases a user can log in without specifying a password:

- If the name of the requesting host is entered in the `/etc/hosts.equiv` file on the remote host and the user is using the same login name.
- If the host name and the login name of the requesting user are listed in the `.rhosts` file in the home directory of the remote user.

If you cannot access the remote host with `rlogin` because your password is not accepted, contact your system administrator.

## 9.2.6 Logging in to a remote host with telnet

`telnet` allows you to log in on hosts which are running under a different operating system, provided these support `telnet`. On UNIX systems you can use `rlogin` or `telnet`.

If you log in to a UNIX system with `telnet`, you must always give your login name and password. With other operating systems the behavior under `telnet` corresponds to the local behavior of the system when a user session is opened and running.

Let us assume, for example, that you want to log in to the UNIX system `mercury`. Enter `telnet` followed by the host name. `telnet` notifies you as soon as the connection to the other host has been established and informs you of your escape character. This is used to interrupt a session. Afterwards you log back into the system in the normal way.

```
venus$ telnet mercury
Trying...
Connected to mercury.
Escape character is '^['.
'UNIX(r) System V Release 4.0 (mercury)
login: hans
Password:
...
```

**Note:**

If you attempt to log in to a host which is not known in your network, `telnet` displays an appropriate message and a prompt (`telnet>`). Terminate `telnet` by entering `quit` or `q`.

**Closing a telnet connection**

To close a connection established via `telnet`, either press the END key or enter the `exit` command.

## 9.2.7 Interrupting a telnet connection

If you are using a shell with job control (`jsh` or `ksh`), you can temporarily interrupt a connection established via `telnet` and resume it again later. To do this, enter the default escape character (normally `[CTRL] -]`). The `telnet>` prompt is displayed and you should enter `z`. This causes `telnet` to run as an interrupted background process.

To reactivate the connection, enter `fg` or `fg %job_number`. `job_number` is the job number of the interrupted process, in this case the login process. If you do not specify a job number, the number of the process that was last placed in the background is used.

```
venus$ telnet mercury
Trying...
Connected to mercury.
Escape character is '^['.
...
mercury$ ([CTRL] -] gives you the telnet> prompt)
telnet> z
Stopped
venus$ fg
telnet mercury (Press Return to call the prompt
 of the remote host)
...
mercury$ exit
Connection closed by foreign host.
venus$
```



### Aborting a telnet connection

As with `rlogin`, you should also only abort a connection established via `telnet` if you cannot terminate it at the end of a session with `exit`.

To abort a `telnet` connection, enter the escape character used by `telnet` (normally `[CTRL]-]`). The `telnet>` prompt is displayed and you enter the `quit` command. This aborts the connection to the other system and you are returned to the local system.

#### Note:

If you log in on several hosts and access each host from a different one, aborting one connection will cause all of them to be aborted, i.e. you will be returned to the local system.

#### Example:

```
venus$ telnet mercury
Trying...
Connected to mercury.
Escape character is '^[]'.
...
mercury$ ([CTRL]-] gives you the telnet> prompt)
telnet> quit
venus$
```

## 9.3 Copying files between hosts

The following methods are available for copying files between hosts:

- using the `Slfmlan` user interface and the options  
*Fetch file from remote computer* and  
*Send file to remote computer*
- using the file transfer program `FTP`
- using the `rcp` command

## 9.3.1 Copying files via Slfmlan

### 9.3.1.1 Fetching files from a remote host

*Fetch file from remote computer* enables you to fetch a file from a remote host. As with the *Login to remote computer* function, you are prompted for the name of the remote host and the user (login) name for the remote host from which you want to fetch a file.

The following input form is displayed:

2 Connect to remote Computer Form

Name of remote computer:  
Login on remote computer:

Once you have made your entries and confirmed them by pressing **SAVE**, an input form is displayed in which you can specify the file you wish to fetch.

3 (ftp) Log on to remote computer

Name of remote folder:  
Name of remote document:

#### *Name of remote folder*

Enter the name of the directory (= folder) containing the file (= document) you want to fetch.

You may enter the following:

- No entry: the remote user's home directory is assumed and the system searches for the file there.
- Relative path name, e.g. `work/letters`

In this case the system looks for the file in the `work/letters` directory in the home directory of the remote user.

- Absolute path name, e.g. `/usr/hans/work/letters`

In this case, the remote host fetches the document from the directory called `letters` in the named path.

*Name of remote document*

Enter the name of the file you want to fetch from the remote host.

Once you have completed your entries and confirmed them by pressing SAVE, the next input form prompts for the name of the target directory into which you want to copy the file.

Use the cursor keys to select the directory (File folder) into which you want the document copied. The greater than character (>) is a flag showing which target you have selected.

```
4 Target Folder/home/frank
>., - File folder
.. - File folder
.hosts -
.sh_history -
.TRASH - File folder
```

Press OPEN-FOL to switch to whichever directory you want to select. A new selection form appears showing the contents of the directory in question. If you fail to select a directory, the system outputs the error message:

```
no folder selected
```

Press SAVE to confirm your choice.

**Caution:**

If a file with the same name already exists in the current directory on the local host, it is overwritten by the incoming file.

The system subsequently prompts you for a password. Once you enter the password, you will receive a message indicating the outcome of the document transfer.

*Example:*

```
230 User frank logged in
200 Type set to I
200 Port command successful.
150 Opening Data connection for
226 Transfer complete
. . .
Press ENTER to continue
```

The line 226 *Transfer complete* tells you that the data transfer was successful. If the attempt was unsuccessful, check the remote directory and the name of the file you specified for the transfer.

### 9.3.1.2 Sending files to a remote host

*Send file to remote computer* enables you to send a file to a remote host. In order to use this function, you need the login name, the password, the name of the directory and the name of the file on the remote host.

You will be prompted for entries in the same sequence as when using the function *Fetch file from remote computer*.

2

Connect to remote Computer Form

Name of remote computer: \_\_\_\_\_

Login on remote computer: \_\_\_\_\_

Enter the name of the remote host and the remote user name. Once you press **SAVE**, you will be prompted for the name of the directory and the name of the document into which you want to transfer the file.

3

(ftp) Log on to remote computer

Name of remote folder: \_\_\_\_\_

Name of remote document: \_\_\_\_\_

Press **SAVE**. The “Source Folder” selection form appears, enabling you to choose the local document you want to send.

4

Source Folder/home/frank

>.

..

.rhosts

.sh\_history

bin

- File folder

- File folder

-

-

- File folder

Confirm your entry by pressing **SAVE**. You will be prompted for a password before the document is transferred.

### 9.3.2 File transfer with ftp

The program `ftp` (File Transfer Protocol) can be used to copy files from one host in the network to another.

`ftp` differs from the `rcp` command in the following respects:

- `ftp` opens an `ftp` session on the remote host. The file is transferred in interactive mode. `ftp` provides a command interface, offering commands for activities such as navigating within the remote file system, creating and deleting files and directories, etc.
- `ftp` enables you to exchange files also between systems which are not running the UNIX operating system.

Once you have started `ftp` with the `ftp` command, you will find yourself in an interactive session with the `ftpd` daemon on the remote host. This daemon is the component of the `ftp` program responsible for handling the file operations on the remote host. As soon as the connection has been established, a message notifies you of this and you are prompted to log in. If you are entered in the password database of the remote host, your login name is displayed and you need only press the Return key. When you have entered the correct password, you can access the files on this host. Below is an example of the login sequence:

```
venus$ ftp dinger
Connected to dinger.
220 dinger FTP server ready.

Name (dinger:hugo):
331 Password required for stone.
Password: (Enter your password here)
230 User stone logged in.
ftp>
```

In the above example, the user logs in to the remote `ftp` server under the same login name (`hugo`) as he is using on the local system. If you want to work on the remote system under a different login name, enter this login in the line `Name ( . . . )`.

#### Anonymous ftp

If you have not been entered in the password database of the remote host, you can still access the files on that host if it has been configured for "anonymous" `ftp`. This is the responsibility of the system administrator.

Opening a session via anonymous ftp could look like this:

```
venus$ ftp mountain
Connected to mountain.
220 mountain FTP server ready.
Name (mountain:stone): anonymous
331 Guest login ok, send ident as password.
Password: (Enter any character string as ID.)
User anonymous logged in.
ftp>
```

The prompt `send ident as password` indicates that no specific password has been set but the system still expects an entry of some sort (e.g. your login name). When the connection has been established, the `ftp>` prompt appears on the screen and you can begin the file transfer.

### 9.3.2.1 Displaying a list of the files on the remote host

As soon as the connection to a remote ftp daemon has been established, you can use the commands `ls` or `dir` to display a list of the files which are accessible on the remote host. You can also change to other directories using `cd`, but you can only transfer the files located in these if you have the necessary authorization.

### 9.3.2.2 Copying files with get and put

The most frequently used ftp commands are `get` and `put`. `get` allows you to copy a file from the remote host to the local host, while `put` allows you to copy a local file to the remote host. Both these commands must be followed by the name of the file to be copied. ftp reports the beginning of the transfer and, when it is complete, displays a message indicating how long it took.

```
ftp> get lab1.results
200 PORT command successful.
150 ASCII data connection for lab1.results (129.144.60.88,1163).
226 ASCII Transfer complete.
local: lab1.results remote: lab1.results
1162 bytes received in 0.08 seconds (14 Kbytes/s)
ftp>
ftp> put lab5.data
200 PORT command successful.
150 ASCII data connection for lab5.data (129.144.60.88,1165).
226 Transfer complete.
local: lab5.data remote: lab5.data
1162 bytes sent in 0.04 seconds (28 Kbytes/s)
ftp>
```

### 9.3.2.3 Copying multiple files with mget and mput

The `mget` and `mput` commands allow you to copy several files at the same time to or from a remote host. The files to be transferred are specified by means of metacharacters. The metacharacter `*` replaces any character string, while `?` replaces any individual character. As with `get` and `put`, `ftp` also informs you here when the transfer begins. Before `ftp` transfers a file, it prompts you to confirm the transfer. Press `y` to start the transfer or `n` to skip the file. When all the specified files have been transferred, the `ftp>` prompt reappears.

*Example:*

```
ftp> mput lab*
mput lab1.results? y
200 PORT command successful.
150 ASCII data connection for lab1.results (129.144.60.88,1180). 226
Transfer complete.
local: lab1.results remote: lab1.results
31 bytes sent in 0.02 seconds (1.5 Kbytes/s)
mput lab2.data? n
mput lab3.results? y
200 PORT command successful.
150 ASCII data connection for lab3.results (129.144.60.88,1181). 226
Transfer complete.
local: lab3.results remote: lab3.results
75 bytes sent in 1e-06 seconds (7.3e+04 Kbytes/s)
ftp>
ftp> mget report?.final
mget report1.final? y
200 PORT command successful.
150 ASCII data connection for report1.final (129.144.60.88,1195). 226
ASCII Transfer complete.
local: report1.final remote: report1.final
2605 bytes received in .44 seconds (5.8 Kbytes/s)
mget report2.final? n
ftp>
```

### 9.3.2.4 Quitting an ftp session

To quit an `ftp` session, enter the `quit` command at the `ftp>` prompt. This terminates the connection to the remote host and you are returned to the shell on the local host.

### 9.3.2.5 Aborting ftp during file transfer

If the remote host fails during a file transfer, you must terminate the transfer with the **DEL** key. The screen displays a corresponding message and the `ftp>` prompt.

## 9.3.3 Copying files between hosts with rcp

The `rcp` program allows you to copy local files to a remote host or to copy files from a remote host to your local host. To be able to access files on the remote host using `rcp`, you need access permission for the login on the remote host. You will have access permission if

- either the name of your host is entered in the `/etc/hosts.equiv` file on the remote host and you are accessing under the same login name
- or the name of your host and your login name are listed in the `.rhosts` file in the home directory of the remote user.

The command is entered in the following format:

```
rcp origin destination
```

where *origin* specifies the host on which the file is located, and *destination* the host to which it is to be copied. The inputs you can use to replace these arguments in the various modes of transfer are explained in the sections that follow.

### Note:

During copying, the copied files and directories receive the owner and group association of the login name of the destination system. The access permissions are retained.

### 9.3.3.1 Copying a remote file to the local host with rcp

To copy a file from a remote host to your own using `rcp`, enter the command in the following format:

```
rcp host_name.login:file directory
```

or

```
rcp login@host_name:file directory
```



For *host\_name* enter the name of the host on which the file to be copied is located. *login* identifies the login name on the remote host. If you do not specify a *login*, the system assumes that the login name on the remote host is the same as that on the local host. *file* identifies the name of the file to be copied, and *directory* the name of the local directory to which the file is to be copied.

Enter the following, for example, to copy the file `new.mem` in the home directory of login `sabine` on host `pluto` into a directory called `memos` under your login on host `venus`:

```
rcp pluto.sabine:new.mem $HOME/memos
```

You can use the normal directory abbreviations used in UNIX (e.g. `$HOME` for your home directory if you are using the Bourne shell, `"."` for the current directory, and `".."` for the superordinate directory).

If the copied file is to be given a different name, enter a *file\_name* after the destination directory. To copy, for example, the file `new.mem` on the host `pluto` to your home directory and at the same time rename it `my.mem`, enter the following:

```
rcp pluto.sabine:new.mem $HOME/my.mem
```

### 9.3.3.2 Copying a local file to a remote host with rcp

To copy a file on your host to another host using `rcp`, use the following command format:

```
rcp file host_name.login:directory
```

or

```
rcp file login@host_name:directory
```

For *file* enter the name of the file to be copied. For *host\_name* enter the name of the host to which the file is to be copied. *login* is the remote login name. If you do not specify *login*, the system assumes that the login name on the remote host is the same as that on the local host. *directory* indicates the directory to which the file is to be copied.

Enter the following, for example, to copy the file `old.mem` in the current directory on your system to the directory `waste` under the login `sabine` on host `pluto`:

```
rcp old.mem pluto.sabine:waste
```

If the file is to be given a different name on the remote host, enter the desired name after the destination directory.

For example:

```
rcp old.mem pluto.sabine:waste/ancient
```

This command copies the file `old.mem` to the current directory `$HOME/waste` on host `pluto` and renames it `ancient`.

### 9.3.3.3 Copying directories with rcp

To copy a directory and its contents from one host to another, enter `rcp` with the option `-r`. The rest of the command syntax is the same as that used when copying files, except that the file names are replaced by directory names.

To copy a directory and its contents from another host to your host, enter the command in the following format:

```
rcp -r host_name.login:directory local_directory
```

For *host\_name* enter the name of the remote host. *login* is the remote login name. If you do not specify *login*, the system assumes that the login name on the remote host is the same as that on the local host. *directory* is the directory to be copied, and *local\_directory* indicates the directory on your host to which the remote directory is to be copied.

To copy a directory and its contents from your system to another, use the following format:

```
rcp -r local_directory host_name.login:directory
```

For *local\_directory* enter the directory to be copied, and for *directory* the name of the directory on the remote host to which it is to be copied.

## 9.3.4 Executing commands on remote hosts

The `rsh` command allows you to execute a single command on a remote host without having to log in on this host. `rsh` stands for `remote shell` (a command interpreter that can execute commands on another system). Use of `rsh` is advisable if you only want to perform one operation on a remote host.

To be able to use `rsh` for executing commands on a remote host, you require access permission for the login name on the remote host. You will have access permission if

- either the name of your host has been entered in the `/etc/hosts.equiv` file on the remote host and you are accessing under the same login name
- or the name of your host and your login name are listed in the `.rhosts` file in the home directory of the remote user.

To execute a command on another host, enter `rsh` followed by the name of the host, the remote login name, and the command to be executed. You do not need to specify the login name if the local and remote login names are identical. If, for example, you want to view the contents of the directory `letters` under the login `hans` on host `mercury`, enter the following:

```
rsh mercury -l hans ls -C letters
```

When you use `rsh` to execute a command on another host, there is no login procedure; instead, `rsh` communicates with a daemon which creates a shell and executes the command on the remote host. The shell used is the one specified for the login name in the password database on the remote host. If this is a Bourne shell, the `.profile` file available for the remote login name is read. If a C shell is created, the system reads the file `cshrc` (if available), and, when executing the command, `rsh` uses all aliases that may have been defined. In the same way as `rlogin` and `rcp`, `rsh` also accesses the password database and the `/etc/hosts.equiv` and `.rhosts` files of the remote host in order to establish whether you have access permission for the remote host or for the login name.

### 9.3.4.1 Using shell metacharacters

In the same way as with `rcp`, all shell metacharacters which are not preceded by an escape character and which are not enclosed in quotes are interpreted by the local shell and not on the remote host.

The same applies to the redirection characters `>`, `<` and `|`. If, for example, you enter the command

```
rsh meadow ls /etc > /tmp/list
```

on the host `oak`, the output of the `ls` command on the host `meadow` is redirected to a file called `/tmp/list` on the host `oak`. If, on the other hand, you enter the command

```
rsh meadow "ls /etc > /tmp/list"
```

the output is instead redirected to a file called `/tmp/list` on the host `meadow`, since in this case the redirection does not take place locally.

### 9.3.4.2 Calling rsh without a command argument

You can also enter the `rsh` command without the `command` argument:

```
rsh host_name
```

In this case `rsh` opens a subshell on the remote host via which you can start commands or programs. It is not possible to start interactive programs. The behavior is similar to that caused by entering

```
rlogin host_name
```

except that on the remote host the `.profile` file is not read and the valid environment is the one defined on the local host (e.g. environment variables `PATH`, `SHELL`, etc.).

### 9.3.4.3 Calling rsh under a different name

You can invoke the `rsh` command by creating a symbolic link between the `/usr/bin/rsh` file and a file with the name of a remote host under the name of this host.

To create, for example, a symbolic link between `rsh` and a file called `meadow`, enter the following command:

```
ln -s /usr/bin/rsh $HOME/bin/meadow
```

If the `$HOME/bin` directory has been entered in your search path (environment variable `PATH`), you can execute a command on host `meadow` by calling `meadow`. You could, for example, view the contents of the `/etc` directory on `meadow` with the following command:

```
meadow ls /etc
```

You can create such links for all remote hosts which you regularly access. Use of the `$HOME/bin` directory is not obligatory; you can create links in any directory for which you have write permission and which is specified in your search path. You can only use these commands for access if you have on all remote hosts a login name which is identical with your local login name and a corresponding entry exists in the `/etc/hosts.equiv` or `$HOME/.rhosts` file.

## 9.4 Obtaining network information

You can use the following functions to obtain information about the LAN to which your host is connected.

### 9.4.1 Listing defined hosts in the network

The *Defined computers* function in the *Slfmlan* user interface lists all the hosts in the LAN that are known to your network:

An example of a typical display format is shown below:

Computer Name   Internet Address   Alias(es)

|        |               |        |
|--------|---------------|--------|
| annex3 | 144.145.16.25 | pttt07 |
| apm01  | 144.145.16.3  |        |
| apm02  | 144.145.16.4  |        |
| apm03  | 144.145.16.5  |        |
| apm04  | 144.145.16.6  |        |

*Computer Name*

The name assigned to each host during installation.

*Internet address*

The address assigned to each host during installation.

*Alias(es)*

A maximum of three aliases can be defined for each host. A host can be addressed by its alias instead of the computer name.

### 9.4.2 Displaying the active sessions in the network

The following functions enable you to view a list of all the active sessions in the local network:

- the *Active Sessions* function in the *Slfmlan* user interface
- the *rw* and *rusers* commands

An example of a typical display format is shown below:

| User  | Comput./Terminal | Session since | last input |
|-------|------------------|---------------|------------|
| mgast | athens:console   | Jun2 10:03    | :13        |
| petra | seattle:tty09    | Jun2 8:47     |            |
| hans  | seattle:tty03    | Jun2 9:12     |            |

*User*

The login name of the user who opened the current session.

*Comput./Terminal*

The name of the host on which the session is running, and the special file via which the terminal is attached. The two are separated by a colon (:).

*Session since*

The date and time at which the session was opened.

*last input*

The time that has elapsed since the last input, expressed in hours:minutes. This field is blank, unless the last input occurred more than one minute previously.

**Note:**

There is no guarantee that this list includes all the active sessions. Only those hosts on which the `rwhod` daemon is active, or for the `rusers` command the `rusersd` daemon, are included.

**9.4.3    Displaying the status of the hosts in the network**

The following functions display the current status of the hosts in the network:

- the *Status of computers* function in the `Slfmlan` user interface
- the `ruptime` command

Here too the messages can be up to one minute old.

An example of a typical display format is shown below:

| Computer | up/down | since    | Active users | Computer Loading   |
|----------|---------|----------|--------------|--------------------|
| athens   | up      | 2+03:10  | 0 users,     | 1.10, 5.20, 20.30  |
| seattle  | up      | 5:30     | 3 users,     | 4.70, 5.30, 0.00   |
| dublin   | down    | 18+12:00 |              |                    |
| boston   | down    | 5+11:03  |              |                    |
| london   | up      | 10:19    | 1 user,      | 7.12, 10.12, 19.60 |
| utah     | up      | 2+13:30  | 3 users,     | 20.00, 32.50, 3.05 |

*Computer*

The name of the host.

*up/down*

Indicates whether the host is switched on or off. *down* means that the host is switched off or its connection to the network is faulty.

*since* indicates how long the host has been up or down, as applicable.

The format is

*days+hours:minutes*.

*Active users*

The number of users currently working on the host.

*Computer Loading*

The average load on the host for the last 5, 10 and 15 minutes expressed as percentages.

**Note**

There is no guarantee that this list includes all the active hosts. Only those hosts on which the `rwhod` daemon is active are included. By default the daemon is not started on Reliant UNIX systems to avoid network and system overload.

**9.4.4 Obtaining information on the Network Information Service**

The Network Information Service NIS keeps information about the network configuration on a central system in the network, the network management master. You can find out, for example, about the accessible hosts in the network, their names and Internet addresses. The following commands allow you to display information from the data kept by the NIS:

- `ypwhich` shows you which network management master is responsible for your local host.
- `ypcat` or `ypmatch` outputs information from the Network Information Service.

`ypcat` supplies all the information in a network management file. This command allows you to search for a specific piece of information. For example, if you want to know the Internet address of the host `seattle`, you can enter:

```
ypcat hosts.byname | grep seattle
```

The command accesses the network management file `hosts.byname` containing all known hosts, and looks for the entry on the host `seattle`:

```
139.34.56.01 seattle
```

The command

```
ypmatch seattle hosts.byname
```

would produce the same entry.

`ypmatch` produces all entries in a network management file which match a specific pattern, in this case *seattle*.

## 9.4.5 Displaying user information with `finger`

The `finger` command displays information on any user whose name you enter in the command line. `finger` does not allow you to call up any information about other systems. When using `finger` you can enter the login name in either long or short format.

The `finger` command displays the following user information:

- Login name
- Login name in long format
- Home directory and login shell
- Time of the last login on the system on which `finger` was invoked
- Time the last mail was received and the last time mail was read
- Terminal name(s) and the time that has elapsed since the last activity.

Below are two (simplified) examples showing the information output by `finger`. When you call the command on your system you may receive slightly different information.



*Example:*

```
venus$ finger moby@see
[see]
Login name: moby In real life: Ishmael Wong
Directory: /home/shipwreck/moby Shell: /usr/bin/sh
On since Nov 14 06:33:41 on console 4 days 14 hours Idle Time
New mail received Wed Nov 18 20:34:02 1987;
unread since Wed Nov 18 16:20:24 1987
venus$ finger Wilhelm Meier
Login name: willi In real life: Wilhelm Meier
Directory: /home/hamburg/willi Shell: /usr/bin/sh
Last login Wed Oct 21 16:16 on tty0 from huber
No unread mail
```

The `finger` command is used above all to determine whether a specific login is still active.

**Note:**

The command can be executed for remote hosts if they are running the `fingerd` daemon. For security reasons this daemon is not started automatically on Reliant UNIX systems.

## 9.4.6 Querying the operating status of a host with ping

If a remote host does not react to your requests, there could be a problem across the whole network, or it could be that the host is down or is not connected to the network. The `ping` command provides an easy way of establishing whether a host is active in the network or not. The command has the following format:

```
ping host [wait_time]
```

where *host* is replaced by the name of the system to be checked. The optional argument *wait\_time* defines how long (in seconds) `ping` is to try to reach the specified host (the default is 20 seconds). For further information, see the description of the `ping` command.

Let us assume that you have entered the following:

```
$ ping mercury
```

If the host is active and able to receive data over the network, the following message will be displayed:

```
mercury is alive
```

If the host `mercury` is not active or cannot receive packages, `ping` issues the following message:

```
no answer from mercury
```

If the host is not known, `ping` issues the following message

```
unknown host mercury
```

If `ping` outputs an error, you should consult your system administrator.

## 9.5 What do I do if ...

This section describes problems that might occur while you are working with this network software. Information is provided on what might be causing the problems and how to deal with the situation.

Only problems that you can recognize and deal with yourself are discussed in this section. If you have any difficulties dealing with a problem, consult your system administrator or network administrator.

The following problems are discussed in this section:

- what to do if a host cannot be accessed
- what to do if you are denied access to a remote host

### 9.5.1 A host cannot be accessed

You want to use the `rlogin` command or another communication function to access a remote host.

**Problem:** You receive the message: `unknown host`

**Cause:** The host name you have specified is not entered in the `/etc/inet/hosts` file on your system. You may have made a typing error. Note that a distinction is made between upper and lower case.

**Solution:** Determine which hosts in your network can be accessed. If NIS is available on your system, you can do this using `ypcat hosts.byname`. Otherwise you will find the host name in the local file `/etc/inet/hosts`.

Find out the correct host name

If the host has not yet been entered, have the network administrator enter the name of the host you want to reach.

Try to set up the connection again.

**Problem:** After a while you receive the message:  
`connection timed out`.

- Cause:** It was not possible to access the host within a predefined period of time. The reasons for this could be that:
- The host is not switched on
  - The host is not currently installed in the network
  - The host is so overloaded that it could not respond within the pre-defined period of time.
- Solution:** Use the `ping` command to determine whether the host is switched on or not.
- If you receive the message `no answer from host_name`, it is either not switched on or it is not connected to the network.
- If the host is reported as being `alive`, try to set up the connection gain.
- If you are still not able to set up a connection, ask your system administrator for help.

## 9.5.2 Access to a remote host is denied

Using `rsh` or `rcp`, you want to access a login name on a remote host.

- Problem:** You receive the message: `no access permission`.
- Cause:** You do not possess the required access permissions at the remote host. You must be entered in the `.rhosts` file of the remote user or in the `/etc/hosts.equiv` file as authorized to access the remote host.
- You do not have read permission for the file you want to copy.
- You do not have write permission for the directory you want to copy.
- Solution:** Ask the user or the system administrator of the remote host to grant you access permission.
- Problem:** You receive the message: `Login incorrect`
- Cause:** You are not authorized to access files on the remote host because the login name you are using is not entered in the password database of this host.

---

# 10 Electronic mail

## 10.1 Introduction

Reliant UNIX provides you with the *mail* and *mailx* commands for communicating with other Reliant UNIX users. These commands offer various options for sending and receiving messages. You can exchange messages with any user who is connected to your network via a Reliant UNIX computer.

You can use either of these commands to send and receive messages, although *mailx* has a wider range of functions than *mail* and is easier to use. See also the *mail* and *mailx* entries in the *Commands* manual [5].

## 10.2 The mail command

The following sections show you how to send and receive messages using the *mail* command.

### 10.2.1 Overview of the mail command

- You should use *mail* to send messages if
  - you only want to enter short messages, since *mail* does not allow you to use any editing functions
  - you want to use *mail* as a message utility in shell scripts
  - you want to send the contents of a file. Although you could use *mailx* for this purpose, *mailx* does not send messages itself but merely passes them on to the *mail* command.
- You should use the *mail* command to read your mail if you receive only a few messages or if the messages you receive are short.

## 10.2.2 Sending messages using mail

### Sending messages to another user

The following example illustrates an easy way of sending a message. You simply enter the *mail* command on the command line, specifying the recipient's name, in this case *nadine*.

```
$ mail nadine
```

The recipient's name may be

- a login name, e.g. *nadine*, *laura*, *cooper* etc., if the recipient is working on your system
- a system name and login name if the recipient is working on another Reliant UNIX system that can exchange data with yours, e.g. *sys2!bob*.

To send a message, first enter the *mail* command and the recipient's name, and then your message. Once you have finished, you send the message by

- entering a period (.) and then pressing , or
- entering the key combination  d, or
- pressing 

at the beginning of a new line.

```
$ mail nadine
Many congratulations on
your new invention. A totally
silent curtain rail is just what
the world needs.
.
$
```

The Reliant UNIX prompt (\$) in the last line shows you that your message has been placed in the message queue and sent off. The message is then placed in the default mailbox. The default mailbox is a file in the */var/mail* directory with the same name as the specified recipient.

### Sending yourself a message

You can send messages to yourself as reminders. Let us assume, for example, that you have the login name *bob* and want to remind yourself to make a telephone call tomorrow. First, you call the *mail* command with your own login name and enter the message text. Then you send the message to yourself:

```
$ mail bob
Don't forget to call Andy at 10.00 today
.
$
```

The next time you log in, at the latest, the following message will appear on your screen informing you that you have mail waiting to be read:

```
you have mail
```

For information on how to display your mail on the screen and process it, and when the system informs you that you have mail, see section *10.2.3 Receiving messages with mail*.

### **Sending messages to several people simultaneously**

To send a message to several people simultaneously, you enter the login name of each recipient on the command line:

```
$ mail andy lucy albert leo
The meeting on Friday at 9:00 has been
postponed until Monday, 10:30.
Your boss
.
$
```

### **Undeliverable mail**

If you specify a nonexistent login name, the *mail* command is unable to deliver your mail. Instead a two-line message appears, informing you that the command cannot be executed and that your mail must therefore be returned. As well as the system and login name of the originator and recipient, the message also tells you why the command was aborted.

Let us assume that you have the login name *bob* and you want to send the following message to the user with the login name *cooper*: *Meet me at 8.00 this evening*. Failing to notice that you have incorrectly entered the login name as *coop* instead of *cooper*, you attempt to send your message. The shell responds by issuing an error message (the text of which may vary depending on which shell you are using):

```
$ mail coop
Meet me at 8.00 this evening
.
mail: Can't send to coop
mail: Return to bob
$
```

The message you receive from the shell in cases like this is stored in a file in the */var/mail* directory. This file has the same name as your own login name, in this case *bob*. The information in this file tells you why the *mail* command was aborted. You can view the contents of the *bob* file by typing *mail* on the command line:

```
$ mail
From bob Mon Jan 23 16:00 1992
Date: Mon Jan 20 11:00:01 GMT 1992
Original-Date: Mon Jan 20 15:59 1992
Not-Delivered-To: qm2sam!coop due to 02 Ambiguous
 Originator/Recipient Name
 ORIGINAL MESSAGE ATTACHED
 (mail: Error # 8 'Invalid recipient')
Content-Length: 87
Content-Type: text
Content-Length: 29
Meet me at 8.00 this evening
?
```

What happens if you send a message to several people, but one of the specified recipient names does not exist?

In this case, the message is sent only to those people whose login name you specified correctly. As the originator, you receive a message informing you that the mail could not be delivered to the nonexistent users. This message is also stored in the file with the login name of the originator in the */var/mail* directory.

## Sending file contents as a message

You want to send the text contained in the *love\_letter* file to the owner of the login name *anna*. Type

```
$ mail anna < love_letter
$
```

*mail* takes the text from the *love\_letter* file and places it in *anna*'s mailbox. Before you send the message, make sure that you have specified the correct recipient. If you specify another login name that the system recognizes, *mail* will deliver your love letter to the owner of that login name. The user with the login name *anne* might be somewhat surprised to receive an unexpected love letter from you.



Table 103 summarizes the *mail* command for sending messages.

| Command summary                                           |                                                                                                                                                                                                       |
|-----------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Command                                                   | Important argument                                                                                                                                                                                    |
| mail                                                      | [sys_name]recipient                                                                                                                                                                                   |
| Function:                                                 | You call <i>mail</i> with one or more login names (and system names, if applicable) to send your message to the specified recipients.                                                                 |
| Argument:                                                 | [sys_name]<br>recipient                                                                                                                                                                               |
|                                                           | To send a message using <i>mail</i> , you must specify at least one login name as the recipient. If the recipient of the message is not on your system, you must also specify the remote system name. |
| Note:                                                     | You send the message either by typing a period at the start of a new line or pressing <code>␣</code> or the key combination <code>CTRL d</code> .                                                     |
| Full syntax:<br><b>mail</b> [_-tw][_-m_msgtype]_recipient |                                                                                                                                                                                                       |

Table 103: Summary of the mail command (send mode)

### 10.2.3 Receiving messages with mail

You can use the *mail* command to display messages on the screen that other users have sent to you. When you log in or return from another shell script and there is mail for you, the following message appears on your screen:

```
you have mail
```

This means that there is at least one message for you in your mailbox, the *login\_name* file in the */var/mail* directory. If you want to display these messages on your screen, enter the *mail* command without any arguments:

```
$ mail
```

The messages are displayed one at a time, beginning with the most recent message to arrive. An example of the *mail* message display is shown below:

```
$ mail
From tommy Tue May 19 15:33 MET 1992
Content-Length: 67

Bob,
The meeting has been postponed indefinitely.
Regards, your boss

?
```

The first two lines of the output, the message header, contain information on the message, including the login name of the originator, the time the message was sent and the length of the message in characters. The message itself appears between the two blank lines. The question mark is the *mail* prompt. The following subsections tell you what entries you can make in response to this prompt.

### Managing incoming mail

If the message text does not all fit on one screen page, you can interrupt the display by pressing the key combination **CTRL** s. When you are ready to continue, press **CTRL** q. After displaying each message, the *mail* command displays a question mark (?) as a prompt and waits for your input. You have various options; you can

- leave the current message in the mailbox and read the next message,
- delete the current message,
- save the current message for future reference, and
- enter a question mark in response to the *mail* prompt (?) to obtain a list of all the available options.

#### *Reading a message without deleting it afterwards*

If you want to read the next message without deleting the current one, press **↓** after the question mark prompt (?). The current message remains in your mailbox and the next one is displayed. Once you have read all the messages in your mailbox, the shell prompt is displayed.

*Reading a message and then deleting it*

To delete a message from your mailbox, enter *d* (for **d**elete) after the question mark prompt (?):

```
? d
```

The next message in the mailbox, if any, is then displayed.

*Saving a message*

To save a message for future reference, you enter *s* (for **s**ave) after the question mark prompt:

```
? s
```

This writes the message to the default mailbox, the *mbx* file in your home directory. If you want to write the message to a different file, you enter the file name after the *s* command. If, for example, you want to write the message to a file called *mailsave* in your current directory, you enter

```
? s mailsave
```

If *mailsave* already exists, the message is added to it. If there is no file with this name, it is created and the message saved to it. You can then use the *ls* command to check whether the new file has been created in your current directory.

You can also save the message to a file in a different directory. For example, you could save the message to the *memo* file in the *project1* directory by entering

```
? s project1/memo
```

When saving messages, you can specify either the relative or the absolute path name. For information on path names, see chapter 3 *The file system*.

*Quitting mail*

If you do not want to read any more messages, enter the following command after the prompt:

```
? q
```

Any messages that you have not yet read are stored in your mailbox until the next time you use the *mail* command. You can abort the display of a message by pressing **DEL**. *mail* then stops displaying messages, outputs the prompt and waits for your input.

Table 104 summarizes the syntax and function of the *mail* command for reading messages.

| Command summary          |                                                                                                                                                  |
|--------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| Command                  |                                                                                                                                                  |
| mail                     |                                                                                                                                                  |
| Function:                | If you do not specify any options, the <i>mail</i> command displays the messages in your mailbox (the system file <i>/var/mail/login_name</i> ). |
| Note:                    | The question mark (?) at the end of the message prompts you for a response.                                                                      |
| Full syntax:             |                                                                                                                                                  |
| mail[_-ehpPqr][_-f_file] |                                                                                                                                                  |

Table 104: Summary of the mail command (read mode)

**Note:**

Two other commands for receiving messages are *notify* and *vacation*. The *notify* command provides a mechanism for notifying recipients, provided that they are currently logged in, of incoming messages. The *vacation* command allows you to answer incoming messages automatically with a prepared message and store incoming messages for future reference. For further information on these commands, see the relevant entries in the *Commands manual* [5].

**Setting the MAILCHECK variable**

The system checks at regular intervals to see if you have received any new messages. The length of the interval depends on the value assigned to the *MAILCHECK* variable. The default value is 600; the value is given in seconds. You check the setting of your *MAILCHECK* variable as follows:

```
$ set | pg
.
MAILCHECK=600
.
```

You can change the value of this and other default variables by modifying your *.profile* file. To find out how to do this, see section 5.4 *Modifying your login environment* in chapter 5 *The shell*.

### 10.2.4 Summary of the mail command options

The *mail* command allows you to specify options that influence how mail is sent and received. You must type a minus sign directly in front of the options you specify on the command line. The *mail* command interprets anything else as a login name or alias of the recipient. The following table summarizes the most important options of the *mail* command, indicating which commands are for read mode (R) and which commands are for send mode (S). A full list of the *mail* command options is given under *mail* in the *Commands* manual [7].

| Command | Options            | Meaning                                                                                                                                                                                                                               |
|---------|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| mail    | -f <i>file</i> (R) | <i>file</i> is the name of the file from which you want <i>mail</i> to read messages.<br><br>If you do not specify -f <i>file</i> , <i>mail</i> reads messages from the default mailbox specified in the \$MAIL environment variable. |
|         | -p (R)             | <i>mail</i> displays the messages consecutively without pausing, starting with the most recent message to arrive.                                                                                                                     |
|         | -P (R)             | <i>mail</i> displays all messages individually, starting with the most recent message to arrive. Each message is displayed with all header lines.                                                                                     |
|         | -t (S)             | <i>mail</i> inserts a line containing the text <i>To: recipient</i> in front of the message text for each specified recipient.                                                                                                        |

Table 105: Summary of the mail command options

## 10.3 The mailx command

### 10.3.1 Summary of the mailx command

The *mailx* command is an enhanced version of the *mail* command. *mailx* provides many options not available in *mail* for sending and receiving mail. Unlike *mail*, *mailx* allows you to

- send messages using tilde commands (known as escapes),
- add a title to a message you want to send,

- use different editors, e.g. *vi*, to process incoming and outgoing messages,
- integrate existing text into your message,
- modify parts of the message header,
- define electronic signatures,
- use message list arguments,
- specify a user-defined startup file called *.mailrc* containing commands that are executed when you call *mailx*,
- specify various options for saving files, and
- issue commands to reply to messages and send copies to other users (for both incoming and outgoing messages).

If you want to customize your *mailx* environment, you should create a startup file called *.mailrc* in your home directory. Section 10.4 *The .mailrc file* describes the variables you can include in your startup file.

### 10.3.2 Sending messages with mailx

The following subsections show you how to

- insert a title in your message,
- edit messages,
- integrate existing text into messages,
- keep a copy of a message you have sent,
- modify the message header,
- define an electronic signature, and
- quit *mailx*.

To send a message to another Reliant UNIX user, you enter the *mailx* command followed by the login name of the recipient (*bob* in the following example). The system switches to send mode and displays the *Subject:* prompt:

```
$ mailx bob
Subject:
```

If you want to finish entering text and send the message off, enter the *~*. tilde escape. The system responds by displaying the string *EOT* (end of text) and the shell prompt (*\$*):

```
$ mailx sms
Subject: Important memo
A decision has been made
to sell off the plant to a group from Hong Kong.
The Icelandic delegation has already left.
Regards, Ben H.
~.
EOT
$
```

### Adding a message title

You can add a title to your message by specifying the option `-s` (for subject) on the command line. For example, entering the command line

```
$ mailx -s "Important memo" sms
```

is the same as entering

```
$ mailx sms
Subject: Important memo
```

You give the message the same title in both cases. If you use the `-s` option to enter a title that contains more than one word, you must enclose the words in quotes.

### Editing a message

In *mailx* send mode, you can call an editor by entering a tilde (~) at the start of a line, followed by a letter (e.g. `v` for `vi`). The following example illustrates how to use the tilde escape:

```
$ mailx sms
Subject: Calling an editor
I want to call my favorite editor vi
using the tilde escape ~v.
```

At this point you notice that you have made a typing mistake in your message text. To correct this mistake, you must now call an editor. Simply enter the tilde escape `~e` or `~v` and press .

If you have not yet modified the settings of the *mailx* variables *EDITOR* and *VISUAL*, `~e` calls *ed* while `~v` calls *vi*. If, when you enter the tilde escape `~e`, *MAXed* is called instead of *ed*, the default setting of the *EDITOR* variable has been changed. You set the *EDITOR* and *VISUAL* variables either in the *.mailrc* startup file or the *.profile* login script. You can also use the *set* command to change the settings of *EDITOR* and *VISUAL* for the current session. For further information, see *set* in the *Commands* manual [5].

If the default setting is still valid for the *VISUAL* variable, `~v` calls the *vi* editor:

```
$ mailx sms
Subject: Calling an editor
I want to call my favorite editor vi
using the tilde escape ~v.
~v
```

Now correct your typing mistake and quit the editor as usual. You now return to send mode, where *mail* prompts you to continue with your message (with the *(continue)* prompt). You can now check your corrected message by entering the tilde escape `~p` which displays the entire message up to the point where you entered `~p`:

```
.
.
~p
Message contains:
To: sms
Subject: Calling an editor
I want to call my favorite editor vi
using the tilde escape ~v.
(continue)
~.
EOT
$
```

### Integrating existing text into your message

The *mailx* command provides you with four ways of integrating text from another source into the message you are creating. You can

- read a file into your message,
- read a message you have received into the message you want to send,
- incorporate the value of a named environment variable into a message, and
- execute a shell command and integrate its output into your message.

The following examples demonstrate the first two functions, which are the most commonly used of the functions listed. For information on the other two functions, see *mailx* in the *Commands* manual [5].



*Reading a file into a message*

The next example illustrates how to read the contents of a file into your message using the tilde escape `~r`. When you enter `~r` and a file name, the system displays the file name and the number of lines and characters the file contains. Because you are still in send mode, you can then continue entering your message.

```
$ mailx catherine
Subject: Sale of plant
This file deals with the subject
of selling off the plant
~r letters/sale
"letters/sale" 10/725
The plant will be sold off tomorrow to
the Icelandic delegation !!!
- Regards, Ben H.
~,
EOT
$
```

When the recipient receives it, the message will contain the text of the *sale* file in the *letters* directory. You can, of course, use the tilde escape `~p` to preview the contents before sending off the message.

*Incorporating a message you have received into a reply*

You also have the option of incorporating a letter already stored in your mailbox into a reply. In the next example, you first use *mailx* to check the contents of the mailbox. Then enter `-m cooper` to send a message to the user with the login name *cooper*. You then fetch message number two (command: `~f2`), entitled *chess*, from the mailbox and incorporate i.e. read it into the new message. *mail* then prompts you to continue entering the message (with the *(continue)* prompt). At this stage, you can either type in more text or quit *mailx* by entering `~q`.

```
$ mailx
mailx version 4.0 Type ? for help.
"/var/mail/windom": 2 messages 1 new
>N 1 abc Tue May 1 08:09 8/155 memo
 2 tp Mon Apr 30 16:57 4/127 chess
? m cooper
Subject: The next chess move
Cooper - your move!
~f 2
Interpolating: 2
(continue)
As you can see, the game is over!
- Windom E.
~,
EOT
?
```

The tilde escape `~m` works in the same way as `~f`, except that it indents the inserted message by one tab stop. For both `~m` and `~f`, you must start in read mode and switch to send mode by entering a command. Other commands that work in this way are described in section 10.3.3 *Receiving messages with mailx*.

### Keeping a copy of messages you send

The *mailx* command provides you with several options for keeping copies of outgoing messages. You can use the tilde escape `~w` and the `-F` option for this purpose without setting any special environment variables.

The tilde escape `~w` followed by a file name writes the message text to the specified file, provided that this file does not already exist:

```
$ mailx anna
Subject: Saving copies
When you want to save a copy of
a message, use the tilde escape ~w.
~w savemail
"savemail" 2/73
~,
EOT
$
```

If you now display the contents of the *savemail* file, your screen will look like this:

```
$ cat savemail
When you want to save a copy of
a message, use the tilde escape ~w.
$
```

However, this method does not save any of the header information.

The `-F` option appends the message text to the end of a file named after the first recipient. If you have used an alias for the recipient(s), the alias is first converted into the corresponding login name(s) and the first login name is used as the file name. If there is already a file with this name in your current directory, the message text is appended to it.

If, however, you specify the `-F` option on the command line, the message header is preserved. You specify the `-F` option in front of the file name involved:

```
$ mailx -F tommy
Subject: Saving
This method appends this message to a
file called tommy in my current directory.
~.
EOT
$
```

You can check the results of your input by displaying the contents of *tommy*:

```
$ cat tommy
From: kol Fri May 8 11:14:45 1992
To: tommy
Subject: Saving
This method appends this message to a
file called tommy in my current directory.
$
```

Modifying parts of the message header

The header of a *mailx* message has four components, as described in the following table.

|         |                                                                                                                                 |
|---------|---------------------------------------------------------------------------------------------------------------------------------|
| Subject | The message title.                                                                                                              |
| To      | The list of recipients.                                                                                                         |
| Cc      | <b>Carbon copy.</b> A list of the people you want to receive a copy of the message.                                             |
| Bcc     | <b>Blind carbon copy.</b> This is also a list of recipients, but each recipient does not see the names of the other recipients. |

Table 106: Components of the message header

If you enter the *mailx* command together with a login name or alias, you switch to send mode and *Subject:* is displayed to prompt you to enter the message title. Once you have entered the subject line and pressed `↵`, *mailx* waits for you to type in the message text itself. You can supplement or modify the header information at any time in send mode using one of the following four tilde escapes:

|                 |                                                                                                                                                                                                                                        |
|-----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>~h</code> | displays all the header fields containing the current subject, recipient, <i>cc</i> list (carbon copy) and <i>bcc</i> list (blind carbon copy). You can change or supplement a current value or accept it by pressing <code>↵</code> . |
| <code>~t</code> | lets you add names to the list of recipients. Names can be either login names or aliases.                                                                                                                                              |
| <code>~c</code> | <code>~c</code> lets you create or add to the <i>cc</i> list. Enter either the login name or the alias of the recipient.                                                                                                               |
| <code>~b</code> | lets you create or add to the <i>bcc</i> list.                                                                                                                                                                                         |

Table 107: Changing the message header

You must enter all tilde escapes at the start of a line. Any additional specifications you make on the same line as a `~t`, `~c` or `~b` command are interpreted as input for the list involved. If you enter a tilde escape on its own, the corresponding header line is displayed and you can use the backspace key to modify it. Any information you enter on the same line as the tilde escape `~h` is ignored.

Signing your messages

You can use the *sign* and *Sign* environment variables to define two different electronic signatures. You call these signatures using the tilde escapes `~a` and `~A` respectively. Let us assume that you want to send the following message entitled *orders* to the user with the login name *cooper*. If you have assigned the value *Windom Earle* to the tilde escape `~A`, your screen will look like this when you enter this command:

```
$ mailx -s orders cooper
Major Briggs will be expecting you at
20.00 in the white hut
~A
Windom Earle
~.
EOT
$
```

The two commands `~a` and `~A` allow you to define two different electronic signatures. However, because the originator's login name appears automatically in the message header, you do not have to identify yourself with a signature.

Quitting mailx (send mode)

Once you have finished writing your message, you can quit *mailx* by typing one of the following commands:

| Tilde escape                            | Meaning                                                                                                                                                                                                                                                                                                                                                                                                                                |
|-----------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ~.                                      | The tilde escape ~. (tilde period) or the key combination <code>CTRL d</code> . These are the two standard ways of quitting send mode; they also send the message. If you were in mailx read mode before activating send mode, you now return to read mode, shown by the question mark prompt (?) you receive after typing this command. If you started in send mode, however, you now return to the shell, shown by the shell prompt. |
| ~q                                      | The tilde escape ~q. This command interrupts your text input and allows you to quit <i>mailx</i> send mode. If you have entered a message text, this command appends it to a file in your <i>home</i> directory called <i>dead.letter</i> .                                                                                                                                                                                            |
| ~x                                      | The tilde escape ~x. This command interrupts your text input. You quit mailx send mode and return to the shell without saving your text.                                                                                                                                                                                                                                                                                               |
| <code>END</code> or <code>CTRL d</code> | You use one of these commands to quit <i>mailx</i> if you are logged in on a remote computer with <i>rlogin</i> . Your connection with the remote computer is not cleared down.                                                                                                                                                                                                                                                        |

Table 108: Quitting mailx

10.3.3 Receiving messages with mailx

*mailx* provides you with over fifty commands that help you manage and edit your incoming mail. The following sections describe the most commonly used commands and arguments:

- the *msglist* arguments
- commands for reading messages
- commands for deleting messages
- commands for saving messages
- commands for replying to messages
- commands for quitting *mailx*

msglist arguments

The *mailx* commands listed above often use a *msglist* argument. This argument provides the command with a list of messages to process. If a command expects a *msglist* argument and you do not specify one, the command is executed for the current message. Make sure that you use the right argument for the context.

In the following example, you first use the *d* command and the argument *1-2* to delete two messages. You then save the messages from the user with the login name *tommy* to a file called *tommymail* using the *s* command:

```
$ mailx
mailx version 4.0 Type ? for help.
"/var/mail/sms": 3 messages 2 new 3 unread
 U 1 jaf Sat May 9 07:55 7/137 test25
> N 2 todd Sat May 9 08:59 9/377 printerinfo
 N 3 has Sat May 9 12:45 29/1214 partyinfo1
 N 4 tommy Sat May 9 12:50 28/1201 partyinfo2
 N 5 tommy Sat May 9 12:55 12/124 course
? d 1-2 [deletes messages 1 and 2]
? s tommy tommymail [saves all messages from user tommy
 to a file called tommymail]

?
```

You can enter a *msglist* argument in one of the following formats:

| Argument       | Meaning                                                               |
|----------------|-----------------------------------------------------------------------|
| <i>n</i>       | Use message number <i>n</i> as the current message.                   |
| <i>^</i>       | The first of the messages not deleted.                                |
| <i>\$</i>      | The last of the messages not deleted.                                 |
| <i>*</i>       | All messages.                                                         |
| <i>n-m</i>     | An inclusive range of message numbers.                                |
| <i>user</i>    | All messages from <i>user</i> .                                       |
| <i>/string</i> | All messages with <i>string</i> in the subject line; case is ignored. |
| <i>:c</i>      | All messages of type <i>c</i> where <i>c</i> may be                   |
| <i>d</i>       | - deleted messages                                                    |
| <i>n</i>       | - new messages                                                        |
| <i>o</i>       | - old messages                                                        |
| <i>r</i>       | - read messages                                                       |
| <i>u</i>       | - unread messages                                                     |

Table 109: msglist arguments

More examples of *msglist* arguments are given in the following subsections.

### Commands for reading messages

When you log in or return from another shell script and there is mail waiting for you, the following message is displayed on your screen:

```
you have mail
```

This means that there is at least one new message in your mailbox. To read the message(s), you must now use either the *mail* command, as discussed earlier, or the *mailx* command. You will now learn how to read your mail using *mailx*.

You simply enter the *mailx* command which takes you to the *mailx* read mode:

```
$ mailx
mailx version 4.0 Type ? for help.
"/var/mail/tommy": 3 messages 3 new
> N 1 rbt Thur Apr 30 14:20 8/190 memo
 N 2 admin Thur Apr 30 15:56 5/84 new printer
 U 3 sms Fri May 1 08:39 64/1574 reorganization
?
```

- The first *mailx* output line displays the version of the *mailx* command used on your system and reminds you that you can obtain help by entering a question mark.
- The second line contains the path name of the file to which the information displayed applies, and tells you the number of messages and their status.
- The rest of the display is header information from the incoming messages. The messages are numbered in sequence; the last one received appears at the bottom of the list. In many cases, there is a status indicator to the left of the number. The status indicator may be N (for **N**ew) or U (for **U**nread). The greater than character (>) points to the current message. Other fields in the header line show the login name of the user who sent the message, the date and time it was sent, the number of lines and characters the message contains and the message subject. The last field remains blank if the originator did not specify a message subject.

Now display messages two and three by entering the *p* command followed by the *msglist* 2-3:

```
$ mailx
mailx version 4.0 Type ? for help.
"/var/mail/tommy": 3 messages 3 new
> N 1 rbt Thur Apr 30 14:20 8/190 memo
 N 2 admin Thur Apr 30 15:56 5/84 new printer
 U 3 sms Fri May 1 08:39 64/1574 reorganization
? p 2-3
```

If you enter a command without specifying a *msglist*, the command is executed for the message indicated by the greater than character (>). Pressing  has the same effect as entering the *p* command (**p**rint) without a *msglist*. Similarly, the message indicated by the greater than character (>) is displayed. To read other messages, you must enter the *p* command followed by the desired message numbers. Some examples of the *print* command are shown below.

|           |                                                   |
|-----------|---------------------------------------------------|
| ?         | Display the current message.                      |
| ? p 2     | Display message number 2.                         |
| ? p 2-5   | Display message numbers 2 through 5.              |
| ? p tommy | Display all messages from the user <i>tommy</i> . |

You could enter *t* (**t**ype) in place of *p* (**p**rint), since these commands have exactly the same effect.

*Displaying the contents of your electronic mailbox*

The *mailx* command lets you view the messages in your mailbox and decide which ones need your immediate attention.

When you call the *mailx* command, it first tells you how many messages there are in your mailbox. If you are connected to the computer over a slow communication line, only 10 message headers are displayed. You can use the following commands to obtain information about the contents of your electronic mailbox:

|         |                                                                |
|---------|----------------------------------------------------------------|
| ? z     | Display the next screen page of header lines.                  |
| ? z-    | Display the previous screen page.                              |
| ? f sms | Display the headers of all messages from the user <i>sms</i> . |



Switching to other mailbox files

If you enter *mailx* without any command-line arguments, the contents of the file */var/mail/login\_name* are displayed on the screen. *login\_name* is your own login name. *mailx* also lets you switch to other mail files and manipulate their contents using any *mailx* command, provided that you have the corresponding permissions.

You change to another file by entering the *fi* or *fold* command, followed by the *file\_name*. These two commands have identical functions. You can also use the following metacharacters in place of the *file\_name* argument:

|                     |                                                                                                       |
|---------------------|-------------------------------------------------------------------------------------------------------|
| %                   | The user' s default mailbox ( <i>/var/mail/login_name</i> ).                                          |
| % <i>login_name</i> | The mailbox of the owner of <i>login_name</i> . You must have the required permissions for this file. |
| #                   | The previous file.                                                                                    |
| &                   | The current mailbox.                                                                                  |

The next example first lists the messages you already have. You then use *fi* & to switch to your mailbox, in this case a file called *sms* in the */var/mail* directory. Since this is your own mailbox, you automatically have the necessary permissions. You quit *mailx* using the *q* command.

```
$ mailx
mailx version 4.0 Type ? for help.
"/var/mail/sms": 3 messages 2 new 3 unread
 U 1 jaf Sat May 9 07:557/137 test25
> N 2 todd Sat May 9 08:599/377 printerinfo
 N 3 has Sat May 9 11:08 29/1214 partyinfo
? fi &
Held 3 messages in /var/mail/sms
"+mbox": 74 messages 10 unread
.
.
.
? q
$
```

Deleting messages

You can delete a message by entering the *d* command followed by a *msglist* argument. If you do not specify a *msglist*, this command deletes the current message. The messages are not deleted until you exit the mailbox file you are currently processing. Until you do, the *u* command (**u**ndelete) gives you the opportunity to change your mind. Once you have issued the *q* command (**q**uit) or changed to another file, however, your message is deleted permanently.

*mailx* permits you to combine the delete and print commands. You enter the *dp* command (**d**elete and **p**rint) to delete the current message and display the next one. The most important delete commands are listed below:

|         |                                                        |
|---------|--------------------------------------------------------|
| ? dp    | deletes the current message and displays the next one. |
| ? d 2-5 | deletes messages 2 through 5.                          |
| ? d :r  | deletes all the messages you have read.                |
| ? d *   | deletes all your own messages.                         |

The following is a practical example of how the delete command works. It illustrates how to delete messages 3 through 5:

```
$ mailx
mailx version 4.0 Type ? for help.
"/var/mail/sms": 3 messages 2 new 3 unread
 U 1 jaf Sat May 9 07:557/137 test25
> N 2 todd Sat May 9 08:599/377 printerinfo
 N 3 has Sat May 9 12:45 29/1214 partyinfo1
 N 4 has Sat May 9 13:36 15/1014 partyinfo2
 N 5 has Sat May 9 14:27 16/1027 partyinfo3
 N 6 has Sat May 9 15:18 16/1114 partyinfo4
? d 3-5
?
```

Commands for saving messages

All messages that you do not delete explicitly are saved when you quit *mailx*. All the messages you have read are written to a file in your home directory called *mbox*. Messages you have not yet read remain in your mailbox (*/var/mail/login\_name*).

There are two forms of the command for saving messages: the lowercase *s* and the uppercase *S*. The uppercase version has the following syntax:

```
S [msglist]
```

The messages you specify in the *msglist* argument are placed in a file in your current directory. This file is named after the login name of the user who sent the first message in the list.

The lowercase version has the following syntax:

```
s [msglist] file_name
```

or simply

```
s
```

The following example first checks whether there is any mail. You then want to save the messages entitled *partyinfo2* and *partyinfo3* together in the *partytime* file:

```
$ mailx
mailx version 4.0 Type ? for help.
"/var/mail/sms": 3 messages 2 new 3 unread
 U 1 jaf Sat May 9 07:55/137 test25
> N 2 todd Sat May 9 08:59/377 printerinfo
 N 3 has Sat May 9 12:45 29/1214 partyinfo1
 N 4 has Sat May 9 13:36 15/1014 partyinfo2
 N 5 has Sat May 9 14:27 16/1027 partyinfo3
 N 6 has Sat May 9 15:18 16/1114 partyinfo4
? s 4-5 partytime
?
```

Make sure that you do not specify a login name as the file name, otherwise *mailx* will issue an error message. If you omit both the *msglist* and the *file\_name*, the mail is placed in the *mbox* file in your home directory.

## Commands for replying to messages

There are two forms of the command for replying to mail: the lowercase *r* and the uppercase *R*. The uppercase version sends your reply to the originator of the message only. The lowercase version sends your reply not only to the originator but also to all other recipients.

When you reply to a message, the subject line of the original message is used as the subject line in your reply, as shown in the following example:

```
$ mailx
mailx version 4.0 Type ? for help.
"/var/mail/sms": 3 messages 2 new 3 unread
 U 1 jaf Wed May 6 07:55 7/137 test25
> N 2 todd Wed May 6 08:59 9/377 requirements
 N 3 cooper Wed May 6 11:08 29/1214 chess
? R 2
To: todd
Subject: Re: requirements
```

Let us assume that the user *todd* also sent the message entitled *requirements* to other people. If you then use the lowercase *r* to reply to this message, the message header looks like this:

```
? r 2
To: todd eg cooper jcb tommy
Subject: Re: requirements
```

### Commands for quitting mailx (receive mode)

You quit *mailx* either by pressing **END** or entering the *q* or *x* command. If you quit *mailx* using **END** or the *q* command, you obtain a summary of what you did with your messages:

```
? q
Saved 1 message in /fs1/tommy/mbox
Held 1 message in /var/mail/tommy
$
```

This shows you that the user *tommy* received at least two messages. He read one of these messages and either did not read the other one or wrote it to the */var/mail/tommy* file using a command. If there were any other messages, they were either deleted or written to other files, and *mailx* does not inform you about them.

If you quit *mailx* using the *x* command, you undo almost all the *mailx* operations you performed. Any mail you read or deleted remains in your mailbox. If, however, you saved messages to other files, that operation is already complete and you cannot undo it using *x*.

### 10.3.4 Summary of the mailx command options

The *mailx* command provides you with options that influence the way in which mail is sent and received. The options you specify on the command line must be preceded by a minus sign. The *mailx* command interprets any other specifications on the command line as the login name or alias of the recipient. The following table summarizes the most important *mailx* command options in read mode (R) and send mode (S):

| Command | Options                | Meaning                                                                                                                                                                                                                                                                                                                                                         |
|---------|------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| mailx   | -f [ <i>file</i> ] (R) | -f specifies the mailbox from which you want <i>mailx</i> to read messages.<br><br><i>file</i> not specified: <i>mailx</i> reads the messages from the user-specific mailbox, <i>\$HOME/mbx</i> .<br><br><i>mailx</i> logs all outgoing messages in a file. The file is named after the first specified recipient and is created in your <i>home</i> directory. |
|         | -F (S)                 | <i>mailx</i> logs all outgoing messages in a file. The file is named after the first specified recipient and is created in your <i>home</i> directory.                                                                                                                                                                                                          |
|         | -H (R)                 | <i>mailx</i> displays only header information and terminates. If there are no messages, the exit status is 0, otherwise 1.                                                                                                                                                                                                                                      |
|         | -n (R,S)               | <i>mailx</i> does not initialize from the global startup file <i>/etc/mail/mailx.rc</i> .                                                                                                                                                                                                                                                                       |
|         | -N (R)                 | <i>mailx</i> suppresses output of the message header lines after the <i>mailx</i> call.                                                                                                                                                                                                                                                                         |
|         | -s <i>subject</i> (S)  | <i>mailx</i> enters <i>subject</i> in the <i>Subject:</i> field of the message header. This allows you add a title to your message.                                                                                                                                                                                                                             |
|         | -u <i>mb</i> (R)       | <i>mailx</i> reads the messages from the default mailbox ( <i>mb</i> ) of the specified user, provided that you have read permission for this file.                                                                                                                                                                                                             |
|         | -V (R)                 | <i>mailx</i> displays its version number and terminates.                                                                                                                                                                                                                                                                                                        |

Table 110: Summary of the mailx command options

## 10.4 The .mailrc file

The *.mailrc* file is a startup file in your home directory that is processed when you call *mailx*. *mailx* first processes the global startup file */etc/mail/mail.rc*, followed by the user-specific startup file *\$HOME/.mailrc*, provided these files exist. These files contain commands to be executed when you call *mailx*. This section describes the environment variables most commonly used in the *.mailrc* file. To set these environment variables for a particular session, you must be in read mode. All the environment variables you can set under *mailx* are listed under *mailx* in the *Commands* manual [5].

There may be a global startup file on your system (*/etc/mail/mail.rc*). The system administrator uses this file to set the global variables that are valid throughout the system. However, the variables that you set in your *.mailrc* file take precedence over the corresponding variables in the global startup file *mail.rc*.

### Note:

Even if there is no global startup file on your system, you can still exchange messages using *mailx*. In this case, *mailx* uses default values stored in various programs of the operating system.

The *.mailrc* file can contain most *mailx* commands. Table 111 lists the commands that you may not include in this file:

| Command    | Meaning                                                                                                  |
|------------|----------------------------------------------------------------------------------------------------------|
| ! or shell | Escape to the shell.                                                                                     |
| Copy       | Save the messages in <i>msglist</i> to a file named after the originator.                                |
| edit       | Call the editor set using the <i>mailx</i> variable <i>EDITOR</i> .                                      |
| visual     | Call the editor set using the <i>mailx</i> variable <i>VISUAL</i> .                                      |
| Followup   | Send a reply to the message originator.                                                                  |
| followup   | Send a reply to the message originator and copy it to all the <i>recipients</i> of the original message. |
| mail       | Switch to send mode.                                                                                     |
| Reply      | Send a reply to the message originator.                                                                  |
| reply      | Send a reply to the message originator and copy it to all the <i>recipients</i> of the original message. |

Table 111: Illegal commands in the *.mailrc* file

You can create your own *.mailrc* file using any editor or copy another user's *.mailrc* file. Here is a sample *.mailrc* file:

```
if r
cd $HOME/mail
endif
set allnet append asksub askcc
autoprint dot
set metoo quiet save showto header
hold keep keepsave
set outfolder
set folder='mail'
set record='outbox'
set crt=24
set EDITOR='/bin/ed'
set sign='Roberts'
set Sign='Jackson Roberts,
Supervisor'
set topline=10
alias fred fjs
alias bob rcm
alias alice ap
alias donna dr
alias pat pat
group robertsgrp fred bob alice mark pat
group accounts robertsgrp donna
```

The above *.mailrc* file contains three very useful commands: *set*, *alias* and *group*.

## The set command

The *set* command allows you to assign a value to an environment variable. This command has the following syntax:

```
set
set name
set name='string'
set name=number
```

If you issue the *set* command without arguments, you obtain a list of all defined variables and their values. The *name* argument refers to an environment variable. You can specify more than one *name* argument after the *set* command. Some variables take an alphanumeric (string) value or a numeric value. String values are enclosed in single quotes.

When you assign a value to an environment variable, e.g. using *HOME=dir*, you are telling the shell how to interpret this variable. However, this type of assignment in the shell does not make the value of the variable available to other Reliant UNIX programs that need to reference environment variables. To



enable other programs to access a variable, you must export it. See *export* in the *Commands* manual [5]. When you export a shell variable, you instruct all programs that access this environment variable to import your value. Some variables such as *EDITOR* and *VISUAL* are not specific to *mailx*; you can specify them as global variables and import them from your shell environment. If a value is set in *.mailrc* for an imported variable, it overrides the imported value. You can cancel a variable setting using the *unset* command, but this command only works on variables that you have set in *.mailrc*. It has no effect on imported variables.

The range of environment variables that you can set in your *.mailrc* file is too wide to be described fully in this chapter. Detailed information on these variables is given under *mailx* in the *Commands* manual [5].

#### *The folder, record and outfolder variables*

Three of the variables in the *.mailrc* file shown above deserve special attention because they show how the messages are organized in directories and files. These variables are *folder*, *record* and *outfolder*. All three are interrelated; they allow you to specify the directories and files in which you want to keep copies of messages.

To assign a value to the *folder* variable, you use the format:

```
set folder='directory'
```

This specifies the directory that you want to contain the default mailbox files. If the directory name does not begin with a slash (/), it is assumed to be relative to *\$HOME*. If *folder* is an exported shell variable, you can also specify file names preceded with a slash (/) in commands that call for a *file\_name* argument. The file name is then expanded to place it in the directory you specified for *folder*.

To assign a value to the *record* variable, you use the format:

```
set record='file_name'
```

This tells *mailx* to save a copy of all outgoing messages in the specified file. The header information is saved together with the message text. This variable is disabled by default.

The *outfolder* variable causes the file in which you store copies of outgoing messages (defined in the *record* variable) to be located in the *folder* directory. To set the variable, you must specify it in a *set* command line. The default value is *nooutfolder*.

### The alias and group commands

In the *.mailrc* file shown above, the two commands *alias* and *group* were used as follows: *alias* was used to assign an alias to a single login name. *group* was used to associate the alias *robertsgrp* with five login names. This enables all five names to be called using a single alias. However, you can use both these commands for exactly the same purposes.

The next group command in the example assigns another group name (*robertsgrp*) and an alias (*donna*) to the group name *accounts*. The group name *accounts* expands to six login names.

### The if-endif command

The sample *.mailrc* file used above also contains an *if-endif* command. In this sample file, this command is issued to change to the *\$HOME/mail* directory to read mail. Here, the user elected to set up a subdirectory to handle incoming mail. The full syntax of the *if-endif* command is:

```
if s | r
 mail_commands
else
 mail_commands
endif
```

The *s* (send) and *r* (receive) allow you to specify the commands you want to be executed according to whether *mail* is called in send mode or read mode.

---

# 11 Archiving

## 11.1 Introduction

When working with Reliant UNIX, you archive your files and directories on off-line storage media such as floppy disks, tape cartridges and magnetic tapes. You do this to ensure that the files and directories are still available in the event of a system failure or to exchange them with other users not connected to your computer or network.

This chapter describes certain applications that involve archiving data on off-line storage media. Specifically, it tells you how to

- back up files or directories to floppy disk, tape cartridge or magnetic tape archive,
- list the contents of a floppy disk, tape cartridge or magnetic tape,
- extract files or directories from an archive on a floppy disk, tape cartridge or magnetic tape, and
- convert the format of your files so that you can use them under Reliant UNIX.

Each section ends with a summary of the commands that are relevant to the applications discussed there.

If you require additional information on floppy disks, tape cartridges, magnetic tapes or the drives associated with these media, refer to chapter *12 Off-line storage devices*.

## 11.2 Archiving files and directories

The *tar* and *cpio* commands enable you to save your files and directories to an off-line storage medium. Both these commands set up what is known as an archive on your storage medium. All the files and directories you specify when you call *tar* or *cpio* are written to the desired storage medium. The command then ends the write operation by placing an end-of-archive mark on the magnetic tape or floppy disk. A floppy disk can contain only one archive, whereas a tape cartridge or magnetic tape can contain several. You can use the *mt* and *tapectl* commands (see chapter *12 Off-line storage devices*) to wind to the next or previous archive on a tape, in order to append new archives to existing ones, for instance.

**Caution:**

If there are several archives on a tape cartridge and you overwrite one of them, all the other archives on this cartridge are no longer readable.

The *tar* and *cpio* commands each use a different format to write to your storage medium. You can only read in your files and directories if you use the same command to extract them as you did to archive them. You should therefore make a note of the format (*tar* or *cpio*) you use to write to each storage medium.

Whether you use *tar* or *cpio* depends on how you want to use the archive. If you want to create an archive for your own purposes, use *tar*. The *tar* command is slightly easier to use than the *cpio* command. If, however, you want to exchange your data with a user working on a different system, you should always use *cpio* for reasons of compatibility. *cpio* permits data exchange between systems with different architectures.

Before you call *tar* or *cpio*, insert a floppy disk, tape cartridge or magnetic tape in the appropriate drive. Remove the write protect mechanism from the storage medium you are using. If you are using floppy disks, make sure that they are formatted. Either use preformatted floppy disks or format your floppy disks using the *format* command (see chapter 12 *Off-line storage devices*).

## 11.2.1 Archiving using the tar command

To archive files and directories on a storage medium using the *tar* command, you must specify the *-c* option (**create**). *tar* then creates an archive on your storage medium.

You use one of the following two formats of the *tar* command to archive your files and directories:

```
tar cvnumber file
```

```
tar cvf special_file file
```

Specifying option *-v* (**verbose**) causes the name and block size of each archived file to be written to standard output.

You use special files to address the drive that is associated with the storage medium you are using (floppy disk, tape cartridge or magnetic tape). There are two different ways of addressing a drive:

- Specify a *number* argument:

This is the short way of addressing a drive. The possible values of *number* and the drive associated with each of these values are defined in the default file `/etc/default/tar`. In this file, ten archive numbers (0 through 9) are assigned to special files that represent storage devices. You can display the contents of this file as follows:

```
$ pg -e /etc/default/tar
.
.
device block size istape
archive0=/dev/rdisk/f03ht 18 1440 n
.
.
.
archive8=/dev/rmt/c0s0 20 150000 y
archive9=/dev/exa 20 2048000 y
$
```

You specify one of the numbers that follow the word *archive* in this file as the *number* argument.

- Specify the option and argument *f special\_file*:

If the `/etc/default/tar` file does not contain a suitable entry, you must address the drive involved by specifying *f special\_file* instead of *number*. *special\_file* is the file that represents the drive you are using. Chapter 12 *Off-line storage devices* lists some of the important special files.

In both cases *file* is the name of the file or directory you want to archive. The file name you specify may also include shell metacharacters (see section 5.2 *The shell command language* in chapter 5 *The shell*). If you specify a directory, *tar* archives it together with all the files and subdirectories it contains.

### Caution:

If there is not enough room on the storage medium to archive your file, *tar* terminates with the following error message:

```
tar: tape write error
```

This means that the last file has not been archived successfully.

The following example illustrates how to archive a directory called *ratatouille* and all its subdirectories onto a tape cartridge. In the */etc/default/tar* file, the special file for a 1/4" tape cartridge drive with automatic rewind is usually the ninth entry and has the archive number 8. You must therefore specify 8 as the

*number*:

```
$ tar cv8 ratatouille
a /ratatouille 0 tape blocks
a /ratatouille/ingredients 0 tape blocks
a /ratatouille/ingredients/courgettes 12 tape blocks
a /ratatouille/ingredients/tomatoes 15 tape blocks
$
```

If your system administrator has modified the */etc/default/tar* file so that it no longer contains an entry for this drive, you must specify the relevant special file explicitly. For example, the special file associated with a 1/4" tape cartridge drive with automatic rewind is */dev/rmt/c0s0* (see chapter 12 *Off-line storage devices*).

```
$ tar cvf /dev/rmt/c0s0 ratatouille
a /ratatouille 0 tape blocks
a /ratatouille/ingredients 0 tape blocks
a /ratatouille/ingredients/courgettes 12 tape blocks
a /ratatouille/ingredients/tomatoes 15 tape blocks
$
```

| Command summary                                                                                                                                                                                                   |                                                                                  |                                                                                                                                                                                                  |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Command                                                                                                                                                                                                           | Important options                                                                | Important arguments                                                                                                                                                                              |
| tar                                                                                                                                                                                                               | c, v, f                                                                          | <i>special_file</i><br><i>number</i><br><i>file</i>                                                                                                                                              |
| Function:                                                                                                                                                                                                         | <i>tar</i> archives files and directories on storage media such as floppy disks. |                                                                                                                                                                                                  |
| Options:                                                                                                                                                                                                          | <div>c<br/>v<br/>f</div>                                                         | <div>Create an archive.<br/>Output status information on each archived file.<br/>Specify a special file to address a drive.</div>                                                                |
| Arguments:                                                                                                                                                                                                        | <div><i>special_file</i><br/><i>number</i><br/><i>file</i></div>                 | <div>Name of the special file for the drive involved.<br/>Address the drive via the corresponding entry in <i>/etc/default/tar</i>.<br/>Name of the file or directory you want to archive.</div> |
| Full syntax:<br><b>tar</b> <b>_c</b> [ <b>vwbL</b> ] <i>number</i> [ <b>_blocking_factor</b> ] <b>_file</b><br><b>tar</b> <b>_c</b> [ <b>vwbL</b> ] <b>_special_file</b> [ <b>_blocking_factor</b> ] <b>_file</b> |                                                                                  |                                                                                                                                                                                                  |

Table 112: Summary of tar command for archiving files and directories

### 11.2.2 Archiving using the cpio command

If you want to archive files or directories using the *cpio* command, you must pass the files or directory contents to *cpio* via standard input. You do this using the *find* command. *find* looks for entries in the file system and writes them to standard output. If you only want to print the names of files contained in a specific directory, you use the following simple form of *find*:

```
find directory -print
```

If you want to archive a directory together with all its subdirectories, you go to the directory involved and call *find* as follows:

```
find . -print
```

This causes *find* to write the relative path names of all files and subdirectories contained in this directory to standard output. You must pass this output to *cpio* using a pipe (*|*). For information on how to use the pipe, see section 5.2.3 *Filters and pipes* in chapter 5 *The shell*.

```
find . -print | cpio -ooption
```

*cpio -o (out)* reads the file names from standard input and writes the files to standard output in a special archive format. To archive these files on an external storage medium, you redirect the output to the special file that represents the storage device you want to use (see chapter 12 *Off-line storage devices*).

The following example illustrates how to archive the *ratatouille* directory and all its subdirectories onto a 3 1/2" floppy disk:

```
$ cd ratatouille
$ find . -print | cpio -oBv -O /dev/rdisk/f03ht
.
ingredients
ingredients/courgettes
ingredients/tomatoes
10 blocks
$
```

You use the *B* option to increase the output block size from 512 to 5120 bytes. It is advisable to use this option when archiving large amounts of data, since it ensures faster access.

If you specify the *v* option (**verbose**), the names of the archived files are written to standard output.

If you specify *-O archive*, *cpio* writes its output to the named *archive*. *archive* is the special file associated with your storage medium.

| Command summary                                                    |                                                                            |                                                                              |
|--------------------------------------------------------------------|----------------------------------------------------------------------------|------------------------------------------------------------------------------|
| Command                                                            | Important options                                                          | Important argument                                                           |
| cpio                                                               | -o, B, v<br>-O                                                             | archive                                                                      |
| Function:                                                          | cpio archives files and directories on storage media such as floppy disks. |                                                                              |
| Options:                                                           | -o                                                                         | Write files and directories to an archive.                                   |
|                                                                    | B                                                                          | Increase the output block size.                                              |
|                                                                    | v                                                                          | Output the names of the archived files.                                      |
|                                                                    | -O                                                                         | Redirect output to <i>archive</i> (e.g. a special file).                     |
| Argument:                                                          | archive                                                                    | The archive to which you want output to be redirected (e.g. a special file). |
| Full syntax:                                                       |                                                                            |                                                                              |
| cpio [-o[aABcLvV]][-Cbuffer_size][-Hheader][--Oarchive][-Mmessage] |                                                                            |                                                                              |

Table 113: Summary of cpio command for archiving files and directories



## 11.3 Listing archive contents

If you want to check what archives there are on your floppy disk or tape and what files these archives contain, you can use *tar* or *cpio* to list the table of contents of the storage medium on standard output. You must use the same command to list the contents of your storage medium as you used to create the archive (either *tar* or *cpio*).

### 11.3.1 Listing archive contents using tar

The *tar* command here has the same syntax as in section 11.2.1 *Archiving using the tar command*, except that you must specify the *-t* option (table of contents). Specifying this option simply causes *tar* to list the archive contents of a storage medium. No files are extracted.

```
tar tvnumber file
```

```
tar tvf special_file file
```

For each file in the archive, *-v* (**verbose**) outputs the file name, permissions, user ID, group ID, size in bytes and creation date.

You must address the drive associated with the storage medium you are using, as explained in section 11.2.1 *Archiving using the tar command*.

If you want to list the contents of several archives on the same magnetic tape or tape cartridge, you must issue the *tar* command once for each archive. In this case, you must ensure that you use a special file without automatic rewind (see chapter 12 *Off-line storage devices*), since *tar* always processes the next archive on the tape.

The following example illustrates how to read the contents of a tape cartridge and list them in detail. Since you know that there are two archives on the tape cartridge, you use the special file without automatic rewind and issue the *tar* command twice. If the */etc/default/tar* file does not contain an entry for the drive you want to use, you must address the special file directly using the *f special\_file* option (see section 11.2.1 *Archiving using the tar command*).

```
$ tar tvf /dev/rmt/c0s0n; tar tvf /dev/rmt/c0s0
tar: blocksize = 12
-rwxr-xr-x 112/100 0 Jan 19 8:51 1992 guide
-rw-r--r-- 112/100 1586 Jan 21 11:13 1992 guide/introduction
-rw-r--r-- 112/100 3098 Jan 24 14:04 1992 guide/chap1
-rw-r--r-- 112/100 2078 Jan 31 05:27 1992 guide/chap2

tar: blocksize = 12
-rwxr-xr-x 112/100 0 Dec 10 12:14 1992 nicole
-rw-r--r-- 112/100 0 Dec 12 15:45 1992 nicole/private
-rw-r--r-- 112/100 1465 Dec 16 18:19 1992 nicole/private/frank
$
```

| Command summary                                                                             |                                                                          |                                                                                                                                                                              |
|---------------------------------------------------------------------------------------------|--------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Command                                                                                     | Important options                                                        | Important argument                                                                                                                                                           |
| tar                                                                                         | t, v,<br>f                                                               | <i>special_file</i><br><i>number</i><br><i>file</i>                                                                                                                          |
| Function:                                                                                   | tar reads the contents of an archive and writes them to standard output. |                                                                                                                                                                              |
| Options:                                                                                    | t<br>v<br>f                                                              | Output table of contents of a storage medium.<br>Output status information on each file.<br>Specify a special file to address a drive.                                       |
| Argument:                                                                                   | <i>special_file</i><br><i>number</i><br><br><i>file</i>                  | The special file for the drive involved.<br>Address the drive via the corresponding entry in <i>/etc/default/tar</i> .<br>Name of the file or directory you want to archive. |
| Full syntax:<br><b>tar _t[vL]_number_._file</b><br><b>tar _t[vfL]_._special_file_._file</b> |                                                                          |                                                                                                                                                                              |

Table 114: Summary of tar command for reading archive contents

### 11.3.2 Listing archive contents using cpio

If you used *cpio* to write to a storage medium, you must also use *cpio*, together with the options *-i* (in) and *-t* (table of contents), to list the contents of this storage medium on standard output. As in the previous section, no files are extracted.

```
cpio -it -I archive
```

The *-I archive* option instructs *cpio* to take its input from *archive*. *archive* is the special file for the storage medium you are using (see chapter 12 *Off-line storage devices*).

The following example illustrates how to list the table of contents of a 3 1/2" floppy disk on standard output.

```
$ cpio -it -I /dev/rdisk/f0
.
cookery
ingredients
ingredients/courgettes
ingredients/tomatoes
10 blocks
$
```

| Command summary                     |                                                                                                |                                                                                                                                                                   |
|-------------------------------------|------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Command                             | Important options                                                                              | Important argument                                                                                                                                                |
| cpio -                              | -i, t,<br>-I                                                                                   | archive                                                                                                                                                           |
| Function:                           | cpio reads the contents of an archive and writes them to standard output. No files are copied. |                                                                                                                                                                   |
| Options:                            | -i<br>t<br>-I                                                                                  | Read a file archive from standard input.<br>Output the table of contents of an archive.<br>Read input from <i>archive</i> . <i>archive</i> may be a special file. |
| Argument:                           | archive                                                                                        | The archive from which you want the command to read its input (e.g. a special file).                                                                              |
| Full syntax:<br>cpio -it -I archive |                                                                                                |                                                                                                                                                                   |

Table 115: Summary of cpio command for reading archive contents

## 11.4 Extracting files and directories from an archive

You can extract files and directories that you have archived on an off-line storage medium at any time. Remember to use the same command to extract your files and directories as you used to archive them since *tar* and *cpio* use different formats.

### 11.4.1 Extracting files and directories using tar

To extract files and directories from an archive on an off-line storage medium, you use *tar* with the *x* (extract) option. This option causes *tar* to extract the specified file or directory from the archive. If you archived the files or directories under relative path names, they are restored in your current directory. If they are archived under absolute path names, they are restored in the appropriate directory, provided that it exists.

The *tar* command has the following syntax when used to extract files and directories:

```
tar xvnumber file
tar xvf special_file file
```

The *v* option lists the following information on each extracted file or directory: *x* (for extract), the name of the file or directory, its size in bytes and its block size. To find out how to address the drive associated with your storage medium, see section 11.2.1 *Archiving using the tar command*. If you specify a directory for *file*, all its files and subdirectories are extracted. If you do not specify a *file*, *tar* extracts all the archive contents.

The following example illustrates how to extract the *nicole/trash* directory from an archive on 3 1/2" floppy disk and read it into your current directory.

```
$ tar xvf /dev/rdisk/f03ht nicole/trash
```

| Command summary                                                                                                                                                                                                            |                                                                                          |                                                                                                                                                                                          |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Command                                                                                                                                                                                                                    | Important options                                                                        | Important argument                                                                                                                                                                       |
| tar                                                                                                                                                                                                                        | t, v,<br>f                                                                               | <i>special_file</i><br><i>number</i><br><i>file</i>                                                                                                                                      |
| Function:                                                                                                                                                                                                                  | <i>tar</i> extracts the specified file or directory from an archive on a storage medium. |                                                                                                                                                                                          |
| Options:                                                                                                                                                                                                                   | t<br><br>v<br><br>f                                                                      | Extract the specified file or directory from an archive.<br>Output status information on each file.<br>Specify a special file to address a drive.                                        |
| Arguments:                                                                                                                                                                                                                 | <i>special_file</i><br><i>number</i><br><br><i>file</i>                                  | The special file that addresses the drive involved. Address the drive via the corresponding entry in <i>/etc/default/tar</i> .<br>The name of the file or directory you want to extract. |
| Full syntax:                                                                                                                                                                                                               |                                                                                          |                                                                                                                                                                                          |
| <b>tar</b> <b>..x</b> [ <b>l</b> <b>m</b> <b>o</b> <b>v</b> <b>w</b> <b>L</b> ] <i>number_file</i><br><b>tar</b> <b>..x</b> [ <b>l</b> <b>m</b> <b>o</b> <b>v</b> <b>w</b> <b>f</b> <b>L</b> ] <i>..special_file..file</i> |                                                                                          |                                                                                                                                                                                          |

Table 116: Summary of tar command for extracting files and directories

### 11.4.2 Extracting files and directories using cpio

If you used *cpio* to archive files and directories on an off-line storage medium, you must also use *cpio*, together with the *i* option (in), to extract the archive. The extracted files and directories are read into your current directory.

`cpio -ioption -I archive pattern`

The *-I archive* option instructs *cpio* to take its input from *archive*. *archive* is the special file associated with the storage medium you are using (see chapter 12 *Off-line storage devices*). *pattern* defines the files or directories you want to extract from the archive, and may include shell metacharacters (see section 5.2 *The shell command language* in chapter 5 *The shell*). If you do not specify a *pattern*, the command extracts all the files and directories contained in the archive.

The following example illustrates how to extract the contents of an archive on tape cartridge.

```
$ cpio -iBdv -I /dev/rmt/c0s0
.
blabla
ingredients
ingredients/courgettes
ingredients/tomatoes
10 blocks
$
```

As in section 11.2.2 Archiving using the cpio command, this example includes additional options. The *B* option increases the block size from 512 to 5120 bytes. You must specify the *d* (**directory**) option if the archive contains directories as well as files. This causes *cpio* to restore these directories when extracting the archive contents. The *v* option causes the command to write the names of the extracted files and directories to standard output.

| Command summary                                                                                                                                                                                                                               |                                                                                 |                                                                                                                                                                                                     |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Command                                                                                                                                                                                                                                       | Important options                                                               | Important argument                                                                                                                                                                                  |
| cpio                                                                                                                                                                                                                                          | -i, B, d, v,<br>-I                                                              | <i>archive<br/>pattern</i>                                                                                                                                                                          |
| Function:                                                                                                                                                                                                                                     | <i>cpio</i> extracts files and directories from an archive on a storage medium. |                                                                                                                                                                                                     |
| Options:                                                                                                                                                                                                                                      | -i<br>B<br>d<br>v<br>-I                                                         | Extract files and directories from an archive.<br>Increase the output block size.<br>Restore directories.<br>List the names of the extracted files.<br>Read input from archive (e.g. special file). |
| Arguments:                                                                                                                                                                                                                                    | <i>archive</i><br><br><i>pattern</i>                                            | The archive from which you want the command to read its input (e.g. special file).<br>Specify which files or directories you want to extract.                                                       |
| Full syntax:<br><b>cpio</b> <i>[-i[bBcdfkmrsSuvV6][<i>[-C</i> <i>buffer_size</i>][<i>[-E</i> <i>file</i>][<i>[-H</i> <i>header</i>][<i>[-I</i> <i>archive</i> [<i>[-M</i> <i>message</i>]]</i> ][ <i>[-R</i> <i>user_ID</i> ][ <i>pattern</i> |                                                                                 |                                                                                                                                                                                                     |

Table 117: Summary of cpio command for extracting files and directories

## 11.5 Conversion

Exchanging data with other users via off-line storage media may involve moving it from one operating system to another. If this is the case, it is not enough simply to write your files (or directories) in Reliant UNIX format to a storage medium using *tar* or *cpio*. Neither can you simply use *tar* or *cpio* to extract files that have a non-Reliant UNIX format. You must first convert your files into the format of the other operating system.

This section describes the two most common situations that require you to convert your files for data exchange purposes.

### 11.5.1 Converting files using the *dd* command

Like most UNIX derivatives, Reliant UNIX is based on the ASCII character set (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange). This character set uses seven bits to represent each character internally; the eighth bit is a parity bit. However, characters may also be represented internally using eight bits, i.e. all eight bits of a byte are used for character coding. Reliant UNIX can also process 8-bit character sets by means of the **N**ative **L**anguage **S**ystem (NLS), described in detail in the *Commands* manual [5].

Some 8-bit character sets, however, must be converted before Reliant UNIX can process them. One of these 8-bit character sets is EBCDIC (**E**xtended **B**inary-**C**oded **D**ecimal **I**nterchange **C**ode), which is used primarily on mainframes. If you want to exchange files between systems that use different character coding, you must convert your files first. Conversion is an important application of the *dd* command.

The *dd* command allows you to convert your files, e.g. from ASCII to EBCDIC and vice versa, and write them to a storage medium. You can also convert files in order to extract them from a storage medium. The general syntax for file conversion using *dd* is:

```
dd if=input_file of=output_file conv=conv_type
```

*if=input\_file* is the file from which you want *dd* to read its input. This may be a Reliant UNIX file you want to convert or a special file via which you want to access a storage medium in order to extract and convert a file. *of=output\_file* is the name of the output file to which you want *dd* to write its output. This may be the name of the converted file or, if you want to write the output to an off-line storage medium, the name of the relevant special file. The option *conv=conv\_type* allows you to specify the type of conversion you want *dd* to carry out.

The following example illustrates how to convert the ASCII file *cleopatra* into an EBCDIC file and write it to a tape cartridge.

```
$ dd if=cleopatra of=/dev/rmt/c0s0 conv=ibm
```

The option *conv=ibm* causes the command to convert the ASCII file into an EBCDIC file using an IBM-specific EBCDIC table.

| Command summary                                                                                                                                     |                     |                                                                                                                                                                                                |
|-----------------------------------------------------------------------------------------------------------------------------------------------------|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Command                                                                                                                                             | Important options   | Important arguments                                                                                                                                                                            |
| dd                                                                                                                                                  | if=<br>of=<br>conv= | <i>input_file</i><br><i>output_file</i><br><i>conv_type</i>                                                                                                                                    |
| Function:                                                                                                                                           | dd converts files.  |                                                                                                                                                                                                |
| Options:                                                                                                                                            | if=                 | if= Specify the file from which you want the command to read its input.                                                                                                                        |
|                                                                                                                                                     | of=                 | Specify the file into which you want the command to write its output.                                                                                                                          |
|                                                                                                                                                     | conv=               | Specify the conversion type.                                                                                                                                                                   |
| Arguments:                                                                                                                                          | <i>input_file</i>   | Input file; may also be a special file.                                                                                                                                                        |
|                                                                                                                                                     | <i>output_file</i>  | Output file; may also be a special file.                                                                                                                                                       |
|                                                                                                                                                     | <i>conv_type</i>    | Conversion type, e.g.:<br><br>ascii Conversion from ASCII to EBCDIC.<br>ebcdi Conversion from EBCDIC to ASCII.<br>c<br>ibm Conversion from ASCII to EBCDIC using an IBM-specific EBCDIC table. |
| Full syntax:                                                                                                                                        |                     |                                                                                                                                                                                                |
| dd[_if=input_file][_of=output_file][_ibs=n][_obs=n][_bs=n][_cbs=n][_files=n][_skip=n][_iseek=n][_oseek=n][_seek=n][_count=n][_conv=conv_type[,...]] |                     |                                                                                                                                                                                                |

Table 118: Summary of dd command for file conversion

### 11.5.2 Copying files between Reliant UNIX and DOS using doscp

The *doscp* command allows you to copy Reliant UNIX files onto a DOS floppy disk or copy DOS files from a DOS floppy disk to Reliant UNIX.  
If the Reliant UNIX file you want to copy contains language-dependent special



characters, you must first convert it using the *dosfilt* command so that it can be read under DOS. Similarly, if a DOS file contains language-dependent special characters, you must convert it using the *sinfilt* command after copying it in so that it can be read under Reliant UNIX.

The *dosfilt* command has the following syntax:

```
dosfilt -d file
```

The *-d* option stands for the German character set. *file* is the name of the file you want to convert.

If you want to make a DOS file readable under Reliant UNIX, you convert it as follows:

```
sinfilt -d file
```

Once again, *-d* stands for the German character set and *file* is the name of the file you want to convert.

If you want to copy one or more Reliant UNIX files onto a DOS floppy disk, use the *doscp* command with the following syntax:

```
doscp sifile a:dospath
```

*sifile* is the name of one or more Reliant UNIX files you want to copy to the DOS floppy disk. *a:* is the name of the floppy disk drive according to DOS conventions. If you have a second floppy disk drive, you address it by specifying *b:*. *dospath* is the directory on the DOS floppy disk to which you want to copy your file(s). If you do not specify a *dospath*, Reliant UNIX copies the specified file(s) to the root directory on the DOS floppy disk. If you want to copy only one Reliant UNIX file, you can rename it by specifying the new file name for *dospath*. The file is then stored under the new name on the floppy disk.

The following example illustrates how to copy a Reliant UNIX file called *hippo* to the *species/marsupials* directory on the DOS floppy disk.

```
$ doscp hippo a:species/marsupials
```

**Note:**

The file names you select for DOS files must conform to the DOS naming conventions. DOS file names may contain a maximum of eight characters, optionally followed by a dot and a file name extension of up to three characters. DOS does not allow you to include all the Reliant UNIX metacharacters in file names, and does not distinguish between uppercase and lowercase.

If you want to copy one or more DOS files from a DOS floppy disk to Reliant UNIX, you use *doscp* with the following syntax:

```
doscp a: dosfile sipath
```

*dosfile* is the name or the path name of the DOS file you want to copy to Reliant UNIX. You can also specify several file names, but always in the format *a: dosfile*. *sipath* is the name of the directory to which you want to copy the DOS file(s). If you want to copy only one DOS file to Reliant UNIX, you can specify a new file name for *sipath*. The file is then copied to your current directory.

The following example illustrates how to copy the DOS files *GAMES/WORM* and *USER/NAMES* to the Reliant UNIX directory */usr/dos*. You must create this directory before copying the files.

```
$ doscp a:games/worm a:user/names /usr/dos
```

**Note:**

Reliant UNIX provides you with more commands for processing DOS files and floppy disks (e.g. *doscat*, *dosformat*, *dosrm*, *dosfilt* and *sinfilt*). For a detailed description of these commands, refer to the relevant entries in the *Commands* manual [5]

| Command summary                                                                                                                                                                      |                                                                                                                                                                                                                                                                                                                                                                                                       |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Command                                                                                                                                                                              | Arguments                                                                                                                                                                                                                                                                                                                                                                                             |
| doscp                                                                                                                                                                                | <i>sifile</i><br><i>a:dospath</i><br><i>a:dosfile</i><br><i>sipath</i>                                                                                                                                                                                                                                                                                                                                |
| Function:                                                                                                                                                                            | <i>doscp</i> copies files between Reliant UNIX and DOS.                                                                                                                                                                                                                                                                                                                                               |
| Arguments:                                                                                                                                                                           | <i>sifile</i> Reliant UNIX file you want to copy to a DOS floppy disk.<br><i>a:dospath</i> DOS directory to which you want to copy the file(s) or new name of the Reliant UNIX file.<br><i>a:dosfile</i> DOS file on DOS floppy disk that you want to copy to Reliant UNIX.<br><i>sipath</i> Reliant UNIX directory to which you want to copy the DOS file(s), or new name for the original DOS file. |
| Full syntax:<br><b>doscp</b> [ <i>_-m</i> ][ <i>_-r</i> ] <i>sifile</i> <b>a:</b> <i>dospath</i><br><b>doscp</b> [ <i>_-m</i> ][ <i>_-r</i> ] <b>a:</b> <i>dosfile</i> <i>sipath</i> |                                                                                                                                                                                                                                                                                                                                                                                                       |

Table 119: Summary of doscp for copying files between Reliant UNIX and DOS

# 11.6 Summary of the archiving and conversion commands

| Command | Important options and arguments                                                                             | Meaning                                                                                                                                                                                                                               |
|---------|-------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| tar     | <i>cnumber</i><br><br><i>cf special_file</i>                                                                | Create an archive on an off-line storage medium; address the drive via the corresponding entry in <i>/etc/default/tar</i> .<br>Create an archive on an off-line storage medium; address the drive via the corresponding special file. |
| tar     | <i>tnumber</i><br><br><i>tf special file</i>                                                                | List archive contents on standard output. Address the drive via the entry in <i>/etc/default/tar</i> .<br>List archive contents on standard output. Address the drive via the corresponding special file.                             |
| tar     | <i>xnumber</i>                                                                                              | Extract files from an archive. Address the drive via the corresponding entry in <i>/etc/default/tar</i> .                                                                                                                             |
| cpio    | <i>-o -O archive</i>                                                                                        | Archive files. <i>archive</i> may be a special file, for example.                                                                                                                                                                     |
| cpio    | <i>-it -l archive</i>                                                                                       | List contents of <i>archive</i> on standard output. <i>archive</i> may be a special file, for example.                                                                                                                                |
| cpio    | <i>-i -l archive</i>                                                                                        | Extract files from <i>archive</i> . <i>archive</i> may be a special file, for example.                                                                                                                                                |
| dd      | <i>if=input_file</i><br><i>of=output_file</i><br><i>conv=ascii</i><br><i>conv=ebcdic</i><br><i>conv=ibm</i> | Convert files.<br>File from which <i>dd</i> reads its input.<br>File to which <i>dd</i> writes its output.<br>Conversion from ASCII to EBCDIC.<br>Conversion from EBCDIC to ASCII.<br>Conversion from ASCII to IBM EBCDIC.            |

Table 120: Summary of commands for archiving and conversion

| Command | Important options and arguments       | Meaning                                                                                                                                                                                   |
|---------|---------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| doscp   | <i>sifile</i><br><br><i>a:dospath</i> | Copy files between Reliant UNIX and DOS systems.<br>Reliant UNIX file you want to copy to a DOS floppy disk.<br>Directory on DOS floppy disk to which you want to copy Reliant UNIX file. |
| doscp   | <i>a:dosfile</i><br><i>sipath</i>     | Copy files between DOS systems and Reliant UNIX.<br>DOS file you want to copy to Reliant UNIX.<br>Reliant UNIX directory to which you want to copy DOS file.                              |

Table 120: Summary of commands for archiving and conversion



---

## 12 Off-line storage devices

This chapter contains basic information on

- off-line storage devices, storage media, and how to use these,
- what a special file is and what special files there are, and
- what Reliant UNIX commands you need in order to access special files and drives.

The chapter ends with a summary of the commands that enable you to work with off-line storage devices.

### 12.1 Introduction

Before starting to archive or extract files, you should already have a basic knowledge of Reliant UNIX and in particular be familiar with the material given in the following chapters of this guide:

- *3 The file system*
- *5 The shell*
- *11 Archiving*

You can use storage devices together with suitable storage media to store, back up or archive your data and extract it whenever you need it. Whether a device is built-in or off-line is irrelevant. Reliant UNIX addresses the devices by means of special files

### 12.2 Types of off-line storage device

What devices you can attach, if any, depends on the expansion capability of your computer. For information, refer to the operating instructions for your computer or ask your system administrator or customer support.

Reliant UNIX-systems support the following devices:

- floppy disk drives
- tape cartridge drives
- CD-ROM drives

- WORM drives
- hard disk drives
- magnetic tape drives

The main differences between these devices are the type, interchangeability and capacity of their storage media, and the method they use to access the storage media.

The way in which devices access storage media may be direct or sequential.

#### *Direct access*

In simple terms, this means that a file stored on a floppy disk is divided into records on the disk according to an addressing system. When you read in the file, this system groups the records together to form the original file. It takes only milliseconds for the system to find the individual addresses. Devices that access data directly are therefore faster than devices that access data *sequentially*.

Floppy disk, CD-ROM, WORM and hard disk drives are direct access devices.

#### *Sequential access*

This means that data is stored in a fixed sequence on the storage medium and can be accessed only in this sequence.

Tape cartridge and magnetic tape drives are sequential access devices.

### **12.2.1 Floppy disk drives**

The most frequently used storage device is the floppy disk drive, which accesses data directly. Nowadays new computers only use 3 1/2" floppy disk (drive)s.

For further information on floppy disk drives, see section 12.3 *Using floppy disks*.



### 12.2.2 Tape cartridge drives

Another easy-to-use storage device is the tape cartridge drive, or streamer. The storage medium for this device is the tape cartridge, or streamer tape.

The tape cartridge drive has superseded the magnetic tape drive. Tape cartridge drives and their storage media are much smaller and easier to handle than magnetic tape drives. In addition, most tape cartridges have a greater storage capacity than magnetic tapes.

One example is the Video 8 tape cartridge. Devices that use Video 8 cartridges (also called Exabyte cartridges or Exatape) are about the same size as ordinary cassette recorders; the cartridges themselves are about the same size as the cassettes used in cassette recorders or Video 8 cameras, hence the name. You can store several Gbytes of data on one of these cartridges. Magnetic tape devices, on the other hand, are at least as large as the CPU of an old PC; the tapes themselves may be up to 10 1/2" in diameter. For example, a magnetic tape with a length of 730 m and a recording density of 2460 bytes/cm has a storage capacity of only 180 Mbytes.

Tape cartridge drives access data sequentially.

Tape cartridges are more suitable than floppy disks for data backup purposes because they have a greater storage capacity. Another advantage of tape cartridges is that they can each hold a number of sequential archives, whereas a floppy disk can hold one archive only (see chapter 11 *Archiving*).

For further information, see section 12.4 *Using tape cartridges*.

### 12.2.3 CD-ROM drives

Like a floppy disk drive, a CD-ROM drive is a direct access storage device. *CD-ROM* stands for **compact disc read-only memory**. Access times are slower than with hard disks. The storage medium for this drive is the CD-ROM, which contains software products.

For information on how to use a CD-ROM drive, refer to the operating instructions accompanying your CD-ROM device.

### 12.2.4 WORM drives

*WORM* stands for **w**rite **o**nce **r**ead **m**any. WORM drives work like CD-ROM drives except that you write data yourself once to the WORM disks. You can then read this data as often as required. WORM drives are used for archiving large quantities of data.

For information on how to use a WORM drive, refer to the documentation accompanying your WORM device.

### 12.2.5 Hard disk drives

A hard disk drive is a direct access storage device with one or more hard disks. It provides faster access than the other direct access media, the floppy disk and the CD-ROM/WORM.

There are different types of storage device, characterized by the way in which the hard disks are fitted into the drive:

- devices with permanently integrated hard disks
- devices with interchangeable hard disks (removable disk storage units)

The hard disk capacity required depends on the quantity of user data and the size of the software products provided.

### 12.2.6 Magnetic tape drives

The magnetic tape drive is the storage device for the oldest type of off-line magnetic storage medium. Like tape cartridge drives, magnetic tape drives are sequential access storage devices.

Magnetic tape drives are briefly mentioned here and are supported by Reliant UNIX for historical reasons, but are nowadays no longer used.

## 12.3 Using floppy disks

This chapter provides you with basic information on using *floppy disks*. The information given is based on 3 1/2" floppy disks and the associated drives, since this is nowadays the most frequently used format.

Reliant UNIX supports the tar and DOS formats as well as the most popular write densities.

For more information on floppy disks and other storage media, see the operating instructions accompanying the storage device or your computer.

### 12.3.1 Handling floppy disks

- Avoid exposure of floppy disks to dust and dirt.
- Never touch the magnetic surface in the opening for the read/write head. On 3 1/2" floppy disks, this opening is protected by a sliding cover.
- Do not bend floppy disks.
- Do not cover them with heavy objects.
- Insert them carefully into the drive.
- Do not use alcohol, thinners, etc. to clean them.
- Do not expose them to sunlight or excessive heat.
- Keep them away from magnetic fields.
- Do not store them below +5° C or above +60° C.
- Always write on the label *before* sticking it onto the floppy disk.
- Do not use an eraser on the label of a floppy disk.

### 12.3.2 Distinguishing features of floppy disks

Floppy disks are distinguished by

- their size,
- the number of writable sides, and
- their format.

Size

The *size* of a floppy disk is expressed in terms of diameter, which may be

- 5 1/4" (obsolete)
- 3 1/2"

Number of writable sides

Floppy disks today are usually double-sided, which means that, in theory, you can write data to both sides. Whether you can actually do so, however, depends on your floppy disk drive. For further information, see the operating instructions accompanying your drive or consult your system administrator.

The number of writable sides may be specified as follows on your floppy disk (or on its packaging):

| Specification                              | Meaning                                                                                                 |
|--------------------------------------------|---------------------------------------------------------------------------------------------------------|
| 1-sided<br>one-sided<br>single-sided<br>SS | Data can be written to one side only.<br>"        "<br>"        "<br>"        "                         |
| 2-sided<br>two-sided<br>double-sided<br>DS | Data can be written to both sides.<br>"        "        "<br>"        "        "<br>"        "        " |

Table 121: The number of writable sides on a floppy disk

Format

A floppy disk must be formatted before you can write data to it.

When a floppy disk is formatted, it is divided into tracks and sectors. Tracks are concentric circles around the center of the floppy disk, and sectors are segments of the circle. The sectors divide each track into an equal number of areas, generally known as blocks. Both tracks and sectors are numbered consecutively; each block is assigned a unique track/sector number pair. This number pair is one of the addresses used for direct access (for information on the different types of access, see section 12.2 *Types of off-line storage device*).

Preformatted floppy disks

Most manufacturers supply preformatted floppy disks. The packaging should indicate how they are formatted, e.g.

DS HD/135 tpi, 2MB, formatted

A (3 1/2") floppy disk formatted in this way has 80 tracks per side, 18 sectors per track, 512 bytes per sector and a real storage capacity of 1440 Kbytes. More information on these values is given under *Storage capacity of floppy disks* below.

If you require a different format, you can reformat a preformatted disk just as you would any other floppy disk using the *format* (tar format) or *dosformat* (DOS format) command and the corresponding special file.

tar format is the classic Reliant UNIX format; DOS format is supported by MS-DOS.

Storage capacity of floppy disks

There is a difference between the potential and real capacity, also called the gross or net capacity, of a floppy disk. Potential capacity is the capacity of an unformatted floppy disk, whereas the real capacity is determined by the floppy disk's size and format. The information given on floppy disks usually refers to the potential capacity.

The following table lists the most common formats and their real storage capacities.

| Size   | tpi | No. of sides | Tracks/ side | Sectors/ track | Bytes/ sector | Density | Real capacity |
|--------|-----|--------------|--------------|----------------|---------------|---------|---------------|
| 3 1/2" | 135 | 2            | 80           | 9              | 512           | DD      | 720 Kbytes    |
| 3 1/2" | 135 | 2            | 80           | 18             | 512           | HD      | 1440 Kbytes   |

Table 122: Storage capacity of floppy disks

*tpi* is the track density of a floppy disk. It is the abbreviation of **tracks per inch**.

The recording density of a floppy disk is measured in bits per inch<sup>2</sup>. A floppy disk may have one of the following densities:

- DD (**d**ouble **d**ensity)
- HD (**h**igh **d**ensity)

The appropriate abbreviation is specified on the floppy disk.

There are two ways of determining the storage capacity of a formatted floppy disk:

- Count the number of bytes as for a normal file using the `wc` command (**w**ord **c**ount) with the `-c` option (**c**haracter); specify the name of an appropriate special file as the file name, e.g.:

```
$ wc -c /dev/rdisk/f03ht
1474560 /dev/rdisk/f03ht
$
```

- Calculate the capacity using one of the following formulae (the character `#` stands for a number):

$$(\text{\#sides} * \text{\#tracks} * \text{\#(sectors/track)}) / 2$$

$$(\text{\#sides} * \text{\#tracks} * \text{\#(sectors/track)} * \text{\#(bytes/sector)}) / 1024$$

The result is the storage capacity in Kbytes.

### 12.3.3 Write protection

3 1/2" floppy disks have write protection in the form of a small square of plastic that you can slide up or down. When the floppy disk is write-protected, you can see a small square gap the same size as the slider.

If you want to protect a 3 1/2" floppy disk, you can use your fingernail to click the slider into the *open* position.

If you want to format a floppy disk or write data to it, you must first click the slider into the *closed* position.

## 12.3.4 Using a floppy disk drive

### Inserting a floppy disk into the drive

When you insert the floppy disk, its upper side should be facing upwards and the opening for the read/write head should be facing the front. Push it into the drive until it clicks into position.

The upper side of a 3 1/2" floppy disk is identified by

- an arrow indicating the insertion direction,
- labels on the large slider over the opening for the read/write head, and
- the bevelled right corner located at the front of the floppy disk as you hold it horizontally.

### Removing a floppy disk from the drive

An access LED next to the floppy disk drive indicates whether the drive is still active. Do not remove the floppy disk from the drive until the LED is no longer lit.

To remove the floppy disk, press the release button. The floppy disk will be ejected slightly so that you can remove it easily.

## 12.3.5 Formatting a floppy disk using the `flformat` command

A floppy disk must be formatted before you can write data to it. *flformat* formats floppy disks in tar format.

```
flformat [option] special_file
```

For *option*, you can specify one or more options that modify the formatting procedure. If, for example, you specify `-V` (**v**erify), the command performs a read/write check on randomly selected sectors after formatting. You can instruct the command to check all sectors by specifying `-E` (**e**xhaustive verify).

*special\_file* is the name of the appropriate special file, depending on the size and desired format of the floppy disk you have inserted. If, for example, you want your 3 1/2" DS/HD floppy disk to have 160 tracks, 18 sectors, 512 bytes/sector and a storage capacity of 1440 Kbytes, you enter the command:

```
$ flformat /dev/rdisk/f03ht
```

For every 10 tracks it formats successfully, *flformat* prints a dot on the screen.

12.3.6 Formatting a floppy disk using the dosformat command

You can also use the *dosformat* command to format floppy disks. If you format a floppy disk using *dosformat*, you can process it using the DOS commands of the Reliant UNIX operating system or on a DOS-based system.

The *dosformat* command has the following general syntax:

```
dosformat[option] special_file
```

For *option*, you can specify one or more options, e.g. -v (for **v**olume label). If you specify this option, *dosformat* asks you to enter a floppy disk name of up to 11 characters. This name is then displayed whenever you use the *dosdir* command to list the contents of the floppy disk.

For the *special\_file* argument, you specify the name of a special file. The following table tells you

- what names you can specify for *dosformat*,
- how these names correspond to floppy disk sizes, and
- how each special file influences the formatting of the floppy disk (tracks, sectors and capacity).

| Floppy disk size | Name of the special file                                                            | Real storage capacity | Tracks/ side | Sectors/ track |
|------------------|-------------------------------------------------------------------------------------|-----------------------|--------------|----------------|
| 3 1/2"           | /dev/at/flop/[r]fx3dt<br>(RM200, 300, 400)<br>/dev/ios0/[r]sfdisk???f3dt<br>(RM600) | 720 Kbytes            | 80           | 9              |
| 3 1/2"           | /dev/at/flop/[r]fx3ht<br>(RM200, 300, 400)<br>/dev/ios0/[r]sfdisk???f3ht<br>(RM600) | 1440 Kbytes           | 80           | 18             |
| 3 1/2"           | /dev/ios0/[r]sfdisk???f3et<br>(RM600)                                               | 2280 KByte            | 80           | 36             |

Table 123: Special files for formatting floppy disks with dosformat



You must replace the character ? in the name of the special file with the number of the SCSI floppy drive connected to a SCSI drive.

| Command summary                                                                                                                             |                                                                        |                                                                              |
|---------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------|------------------------------------------------------------------------------|
| Command                                                                                                                                     | Important options                                                      | Important argument                                                           |
| format                                                                                                                                      | -V, -E                                                                 | <i>special_file</i>                                                          |
| Function:                                                                                                                                   | <i>format</i> prepares a floppy disk so that you can write data to it. |                                                                              |
| Options:                                                                                                                                    | -V                                                                     | Performs a read/write check on randomly selected sectors of the floppy disk. |
|                                                                                                                                             | -E                                                                     | Performs a read/write check on all sectors of the floppy disk.               |
| Argument:                                                                                                                                   | <i>special_file</i>                                                    | Name of the special file for the drive involved.                             |
| Full syntax:                                                                                                                                |                                                                        |                                                                              |
| <b>ffformat</b> [ <i>_-f_track1</i> ][ <i>_-l_trackn</i> ][ <i>_-i_interleave-faktor</i> ][ <i>_-V</i> ][ <i>_-E</i> ] <i>_special_file</i> |                                                                        |                                                                              |

Table 124: Summary of the floppy disk formatting commands: format

| Command summary                                                                  |                                                                                                                                                                                                                                       |                                                             |
|----------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------|
| Command                                                                          | Important options                                                                                                                                                                                                                     | Important argument                                          |
| dosformat                                                                        | -v                                                                                                                                                                                                                                    | <i>special_file</i>                                         |
| Function:                                                                        | <i>dosformat</i> prepares a floppy disk so that you can write data to it. If you format a floppy disk with <i>dosformat</i> , you can process it using DOS commands under the Reliant UNIX operating system and on DOS-based systems. |                                                             |
| Options:                                                                         | -v                                                                                                                                                                                                                                    | Assign a volume name to the floppy disk after formatting it |
| Argument:                                                                        | <i>special_file</i>                                                                                                                                                                                                                   | Name of the special file for the drive involved.            |
| Full syntax:                                                                     |                                                                                                                                                                                                                                       |                                                             |
| <b>dosformat</b> [ <i>_-f</i> ][ <i>_-q</i> ][ <i>_-v</i> ] <i>_special_file</i> |                                                                                                                                                                                                                                       |                                                             |

Table 125: Summary of the floppy disk formatting commands: dosformat

## 12.4 Using tape cartridges

This section provides you with basic information on using the *tape cartridge* storage medium.

For detailed descriptions of the *mt* and *tapectl* commands introduced in this section, see the *Commands* manual [5].

### 12.4.1 Handling tape cartridges

Although tape cartridges are much less sensitive than floppy disks due to their rigid protective casing and outer container, you should observe the following rules when handling them:

- Always store tape cartridges in their containers.
- Never touch the tape itself.
- Do not expose tape cartridges to sunlight or excessive heat.
- Keep them away from magnetic fields.
- Do not use alcohol, thinners, etc. to clean them.

### 12.4.2 Write protection

All tape cartridges have a mechanism that allows you to prevent the stored data from being overwritten.

#### *Write protection on 1/4" tape cartridges*

Write protection on 1/4" tape cartridges consists of a revolving lock with an arrow and the word **SAFE** imprinted on the upper side of the cartridge.

To enable write protection on your cartridge, you use an object such as a paper clip to turn the lock until the arrow points to **SAFE**.

If you want to write data to the cartridge, you turn the lock so that the arrow is pointing in the opposite direction.

#### *Write protection on Video 8 tape cartridges*

Write protection on these cartridges consists of a small red slider with *rec* and *save* imprinted on the side.

To enable write protection on your cartridge, you can use your fingernail to move the slider to the *save* position.

If you want to write data to the cartridge, you move the slider to the *rec* position (**record**).

### 12.4.3 Using a tape cartridge drive

For information on inserting and removing tape cartridges, see the operating instructions for your tape cartridge drive. Depending on the type of drive you are using, an LED may light up when a tape cartridge is inserted or accessed.

### 12.4.4 Processing tape cartridges using the *mt* command

You can create a series of archives on a tape cartridge. If, however, you only want to extract the third archive, for example, you must wind the tape in your cartridge to the archive you want to address. You may also want to create new archives on the same cartridge without overwriting the existing ones. Likewise, you must wind the tape on to a specific position. If your device does not retension the tape automatically before the first access, you may want to do this yourself in order to protect your data. You can use the *mt* command (**m**agnetic **t**ape control) to perform these and other tasks.

This command has the following general syntax:

```
mt[-f special_file] command [n]
```

You should specify the *-f* option with the *special\_file* argument if you frequently need to address a different special file. Otherwise, it is advisable to assign the name of a special file to the *TAPE* environment variable. If you then call the *mt* command without *-f*, the special file you have set is addressed automatically.

*command* is the action you want to execute on your tape cartridge. You may not specify more than one action for *command*.

Argument *n* is the number of times you want the action to be repeated.

## EXAMPLES

### *Positioning the tape on an archive*

Let us assume that you have created 4 archives on a cartridge and want to list the table of contents for the third archive only. First, you position the tape at the start of the third archive using *fsf* (forward skip files). You must execute this action twice, since *fsf* merely winds the tape to the end of the current archive. You should use a special file that does not cause the tape to be rewound. Your command line will look like this:

```
$ mt -f /dev/rmt/c0s0n fsf 2
```

Then you list the table of contents of the archive with the same command you used to create the archive, e.g. *tar*. Again, you should use a special file that does not cause the tape to be rewound.

### *Erasing and rewinding the tape*

You are now at the end of the third archive and want to delete the fourth. You then want to rewind the tape to the beginning. To delete an archive, you specify the *erase* function.

#### **Caution:**

On 1/4" tape cartridges, *erase* erases the entire tape.

On Video 8 cartridges and magnetic tapes, *erase* erases the data from the current position to the end of the tape.

To rewind the tape, you specify an appropriate special file. Your command line will look like this:

```
$ mt -f /dev/rmt/c0s0 erase
```

### *Rewinding to the start of the tape*

If you want to rewind your cartridge, you specify the *rewind* action with *mt*. Your command line will look like this:

```
$ mt -f /dev/rmt/c0s0 rewind
```

| Command summary                  |                                                                                                                                                                                                   |                                                                                                                                                                                                                                  |
|----------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Command                          | Option                                                                                                                                                                                            | Arguments                                                                                                                                                                                                                        |
| mt                               | -f                                                                                                                                                                                                | special_file<br>command<br>n                                                                                                                                                                                                     |
| Function:                        | mt winds or positions the tape of a tape cartridge, retensions the tape, writes start-of-archive and end-of-archive marks to it, erases it or provides status information on the addressed drive. |                                                                                                                                                                                                                                  |
| Option:                          | -f                                                                                                                                                                                                | Specify the special file to be addressed.                                                                                                                                                                                        |
| Arguments:                       | special_file                                                                                                                                                                                      | Name of the special file for the drive you want to address.                                                                                                                                                                      |
|                                  | command                                                                                                                                                                                           | Action to be executed, e.g.                                                                                                                                                                                                      |
|                                  | fsf                                                                                                                                                                                               | Position the tape at the start of the next archive.                                                                                                                                                                              |
|                                  | erase                                                                                                                                                                                             | Erase the data from the current position to the end of the tape. On 1/4" cartridges, this erases the entire tape; on Video 8 cartridges and magnetic tapes, it erases the data from the current position to the end of the tape. |
|                                  | rewind                                                                                                                                                                                            | Rewind to the start of the tape.                                                                                                                                                                                                 |
|                                  | n                                                                                                                                                                                                 | Number of times you want an action to be executed.                                                                                                                                                                               |
| Full syntax:                     |                                                                                                                                                                                                   |                                                                                                                                                                                                                                  |
| mt[_-f_special_file]_command[_n] |                                                                                                                                                                                                   |                                                                                                                                                                                                                                  |

Table 126: Summary of the tape cartridge processing commands: mt

## 12.5 Devices and special files

Information on special files can be found under `/etc/default/tar`, while information on devices is provided by the *gettypes* command.

### 12.5.1 Displaying SCSI devices: the *gettypes* command

The *gettypes* command is used to display SCSI devices and some of their parameters. For further details, see under *gettypes* in the *Commands* manual [5].

This command has the following general format:

```
gettypes[option]
```

Options:

- `-R` SCSI devices for the relevant release
- `-A class` SCSI devices for the current configuration

The following can be specified for *class*:

cdrom, dat, disk, floppy, jukebox, raid, rod, smc, tape, videotape, worm, all.

*gettypes* enables you to display on the one hand the SCSI devices that have been freed for the relevant release, and on the other the SCSI devices which are currently configured on the system. You can select particular device types by specifying classes.

Option `-R` outputs the following: the device class, followed by the name of the device, followed by the capacity if this specification is possible (it is not possible for devices with removable media, in which case a 0 is specified). For hard disks the capacity, cylinders and capacity/cylinder are displayed, and for jukeboxes the number of slots (min. and max.), number of drives (min. and max.) and the type of drive are displayed.

Option `-A` outputs the following: the name of the device as it appears under the `autoconf(8)` command, device class, maximum capacity (if possible), path name with suitable options (for device class “tape”), and alias (if one exists).

If you call *gettypes* without options, a formatted list of all devices currently configured in the system is displayed. The information displayed is the same as that provided when you call *gettypes* with the option `-A all`.

| Command summary                                                  |                                                                     |                                                                                                                                                                                       |
|------------------------------------------------------------------|---------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Command                                                          | Important options                                                   | Important arguments                                                                                                                                                                   |
| gettypes                                                         | -R<br>-A                                                            | <i>class</i>                                                                                                                                                                          |
| Function:                                                        | <i>gettypes</i> displays SCSI devices and some of their parameters. |                                                                                                                                                                                       |
| Options:                                                         | -R                                                                  | SCSI devices for the relevant release.                                                                                                                                                |
|                                                                  | -A <i>class</i>                                                     | SCSI devices for the current configuration.<br>The following can be specified for <i>class</i> :<br>cdrom, dat, disk, floppy, jukebox, raid, rod, smc,<br>tape, videotape, worm, all. |
| Full syntax:<br><b>gettypes</b> [ <i>-R</i> ][ <i>-A class</i> ] |                                                                     |                                                                                                                                                                                       |

Table 127: Displaying SCSI devices with gettypes





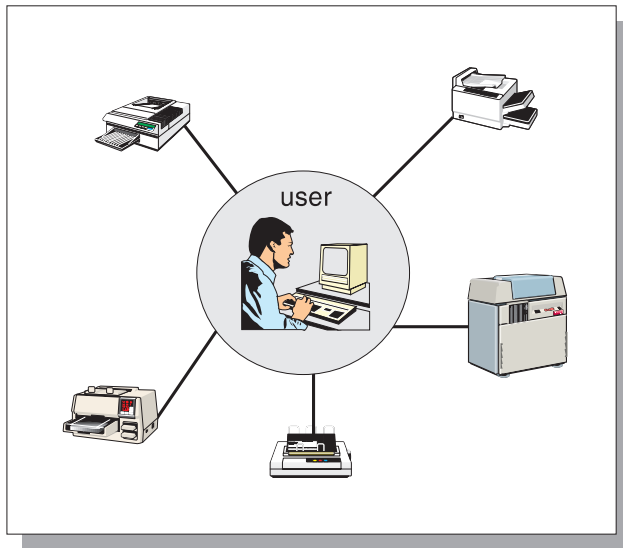
---

## 13 Printing with Spool

This chapter describes the concepts of Spool, which you must be familiar with in order to use this product. It describes the objects which comprise Spool and how these objects interact.

### 13.1 Spool from the user's point of view

From the point of view of the user, Spool represents a number of commands or a menu interface used for printing files. For the user, the entire Spool system consists of a number of printers and printer groups on which documents can be printed (see figure below).

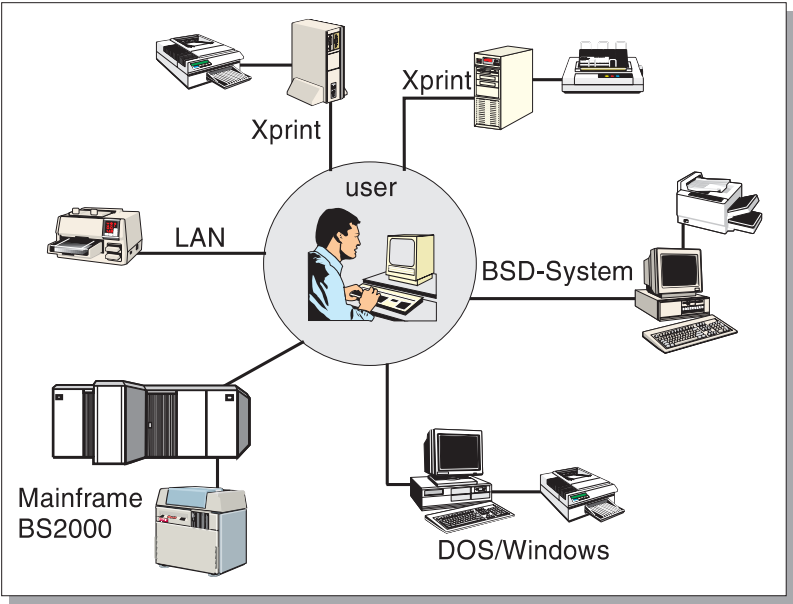


As a user, the first things you deal with are the local workstation on which you are working, and the printers of the Spool system. You can use any of the visible printers for printing your documents. It might be the case, however, that you cannot access some of the visible printers because you have not been granted the necessary rights for using them.

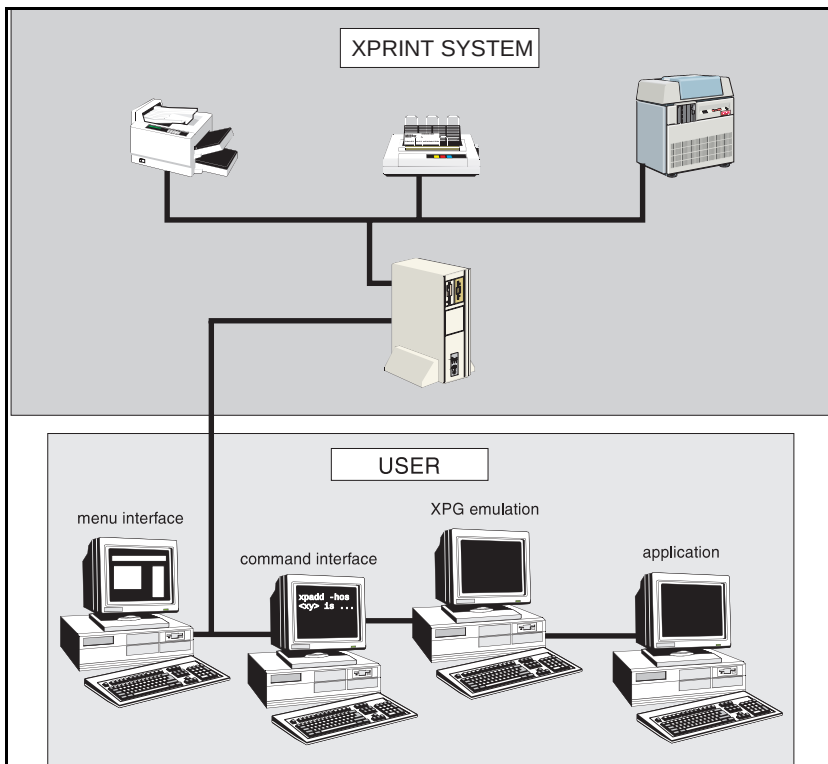
There might also be printer groups available. These printer groups combine several printers to provide quicker spoolout. When printed via a printer group, the print jobs are distributed evenly over the printers in the group so that they can be processed quickly even when print loads are heavy.

In a network, Spool is completely transparent. This means that it does not matter whether a printer is connected to your host or to another host in the network. You can use the printers just as if they were connected directly to your host.

It is even possible to send print jobs to printers outside your own Spool domain, provided that the required gateways are configured. In addition to printers of other Spool domains, you can also use printers which are not connected to a UNIX system with Spool, but to computers running BSD-UNIX, MS-DOS/MS-Windows or BS2000 (see following figure).



As a Spool user you can also decide which interface you want to use on your system. See the following figure for your options:



The various interfaces offer you the following options:

- The menu interface allows you to perform all actions via a graphical user interface.
- Using the command line interface, you can perform all actions by entering Spool commands
- The XPG emulation allows the use of the X/Open commands *lp*, *lpstat*, *cancel*.
- The application programming interface API enables applications to access the Spool system via C function calls.

## 13.2 Spool elements

This section describes the elements that comprise Spool and the possibilities Spool offers its users.

### 13.2.1 Spool objects

Spool is an object-oriented system. This is also reflected in the way it is used. Almost all the Spool commands can be used with the different object types, which is why the number of Spool commands is relatively low. The following section provides an overview of the available objects and their uses.

The following objects are defined in Spool. The parentheses contain the short name of the Spool objects for use in Spool commands:

- The Spool system (-sys)  
The **Spool system** refers to all the Spool objects which are combined in a local administrative unit, a so-called **Spool domain**. Within the Spool system the **Spool administrator** is allowed to perform all the administrative tasks.
- Gateways (-gtw)  
**Gateways** allow you to make a connection from one Spool domain to another or to other spooler systems. Via these gateways you can use the printers and printer groups of the other system involved.

Gateways to DPRINT (spooler system for BS2000 systems) or BSD spooler systems (spooler system of Berkeley BSD UNIX systems) allow users of the local Spool domain to access the printers and printer groups in these spooler systems. Vice versa, it is also possible to access the printers and printer groups of the local Spool domain from DPRINT and BSD spooler systems.

#### Note:

Exchanging data via gateways requires that the local gateway be defined on a Reliant UNIX platform which has FT-SINIX and FTAC-SINIX installed.

- Hosts (*-hos*)

Under Spool the term **host** refers to every computer defined in Spool domain. Hosts are divided into different classes: master, slave and parasite hosts. The difference lies in the way they handle the Spool database.

The **Spool database** contains all the data Spool needs to access Spool resources. This includes information on the defined hosts, users, printers and printer groups as well as on forms and all the other Spool objects. Only the print jobs are stored on the server which is responsible for their execution.

- Servers (*-srv*)

Servers are the Spool objects which administer incoming print jobs and are responsible for scheduling them. Each **server** administers its own print queue which contains all the print jobs for the printers whose **supervisors** are controlled by the server.

- Supervisors (*-spv*)

A **supervisor** is directly responsible for controlling one or more printers. It obtains the job data from the superordinate server and forwards it to the backend program. One supervisor may control several local printers.

- Printer groups (*-dgr*)

Printers can be combined to form so-called **printer groups** to improve the loading distribution of all the printers in the group. If a print job is sent to a printer group, it is printed on the first printer to become available in this group.

- Printers (*-dev*)

Spool can use printers of many different types. The printers are controlled by a **supervisor** and assigned tools, so-called backends. Also, a **PCL** is assigned to the printer.

- Print jobs (*-job*)

**Print jobs** are submitted by users and, as described in the previous section, can be manipulated. They are administered by the **servers** responsible for the respective print queues.

In addition to the normal manipulation options, you can redirect all print jobs for a printer to another printer.

- Forms (*-frm*)

**Forms** are a convenient way of influencing the appearance of **print jobs**. Forms can be used to define paper and page formats, the layup (portrait or landscape), font, appearance of the information pages and user privileges. Spool differentiates between automatic and manual forms. Jobs using

**automatic forms** can be printed at any time: these forms are mounted on the printer by default and usually only define the hopper or the page format (margins) or the font. If a specific form requires that special paper be mounted in the printer manually or that a font cassette be changed, a **manual form** ensures that the job is only printed when the printer has been prepared.

- PCLs (*-pcl*)  
**PCLs (printer capability lists)** are used to define what printing options a printer provides. Furthermore, the PCL contains information on how the options specified at job submission can be converted to printer commands. Spool is supplied with a number of PCLs for the most common printers, and with one generic PCL which in most cases also allows printing on unknown printers.

- Users (*-usr*)  
Spool differentiates between two types of user: **Reliant UNIX users** who are not defined in the Spool system and **Spool users** who are defined in the Spool system. Reliant UNIX users are all users of the Reliant UNIX system on which Spool is running; the default settings for Spool apply to them. If these are not enough, the Reliant UNIX users can be defined in the Spool system.

The administrator can make more settings for Spool users than for Reliant UNIX users:

- Spool users can access their print jobs from all the hosts for which they are registered.
  - They can be assigned a **security level** and thus be granted access to printers with a corresponding security level.
  - **Print quotas** can be defined for the number of jobs or print pages allowed, thus preventing users from making an unlimited number of printouts.
- Filters (*-flt*)  
**Filters** have the function of converting the data sent to the printer by the user into the format required by the printer. For example, regular ASCII files can be printed on PostScript printers because Spool is supplied with the filter `vppost`. Two types of filter are distinguished under Spool: slow filters, which are executed before the Spool process, and fast filters, which are used to provide a link to objects other than the standard components.

- **Fonts (-fnt)**  
Font types supported by the printers in use can be easily be accessed through the definition of fonts.
- **Job templates (-jtp)**  
**Job templates** serve to simplify the use of frequent print commands by allowing the options of the *xpadd* command, which must normally be specified individually, to be combined in groups. These can then be specified via the job template.
- **Permission lists (-prm)**  
**Permission lists** make it possible to define who may use which Spool resources by granting the relevant privileges. The administrator can define the following privileges for users under Spool:
  - Which users may/may not use a printer or a printer group in Spool or other environments.
  - Which users may/may not use a specific form.
  - Which forms may/may not be used on a specific printer.
- **Recovery rules (-rec)**  
**Recovery rules** are used to describe how specific printer conditions are handled.

The default recovery procedure in the case of a printer error can be overridden by a recovery rule. In the case of the printer error “paper\_end”, for example, the Spool system tries by default to print the job until the paper supply is refilled. By means of a recovery rule you could define a different procedure (to suspend the print job and set the printer to NOT\_ACTIVE until the administrator intervenes, for example).

### 13.2.2 Options for printing documents

In the network as well as on a stand-alone system, Spool provides extensive functionality for printing documents (see also the Reference Manual):

- **Selecting a printer of your choice**  
You can print your documents on any printer in the Spool system which you are allowed to use. You can choose either an individual printer or a printer group as the destination. You can also define your personal default printer to which every print job is sent for which you do not explicitly specify a destination, and you can access printers which are not available in your own Spool domain via so-called gateways.

- Automatic adjustment of the document type  
Spool allows you to send any document to any printer. If the relevant filters are defined, the document type is automatically adjusted to the printer type. This makes it possible, for example, to print normal ASCII files on PostScript printers (the filter for this conversion is supplied with Spool).
- Printing several copies of a document  
For every print job you can specify how many copies of the entire job are to be printed. Provided the printer supports this option, you can also print several copies of each individual page. In this case, each page is printed several times before the next page is printed.
- Printing parts of a document (not available for POSTSCRIPT documents)  
Provided the selected printer supports this option, you can also print parts of a document. Here it is possible to define the part to be printed by specifying a range of pages or strings ("print from 'abc' to 'xyz'").
- Printing at specific times  
For every print job, you can specify a time when it is to be printed. This allows you, for example, to schedule longer print jobs for nighttime printing. In addition, you can define that print jobs which have not been printed by a specific time are deleted automatically.
- Printing at a low priority  
Spool allows priorities to be assigned to print jobs. These priorities are user-specific, which enables print jobs of specific users to be given preference over those of others. Less important jobs can be assigned a lower priority.
- Printing confidential documents  
Every user and every printer under Spool has a so-called security level. This allows you to ensure that confidential jobs are only printed on those printers which feature a corresponding security level.
- Selecting forms  
If there are several forms defined for a printer, you can specify which of these forms you want to use for printing. Forms define the page format and the font attributes of a printout. However, you can always change the specifications for an existing form to make it meet your specific needs, for example by specifying a different typeface for a specific job. This also applies if only the default form is defined.
- Notification of the result  
Spool allows you to define how you want to be notified of the status of your print jobs. You can define, for example, that in the event of an error you want user Brown to be notified by mail.



- **Direct printer control**  
You can also control a specific printer directly and define specific printer settings, such as the line spacing.
- **Defaults for print jobs**  
Spool allows you to combine all the options mentioned above in so-called job templates. In this way, you can define frequently recurring settings once and then retrieve them whenever required. These job templates can be defined by each user.
- **Individual, national-language user interface**  
By default, Spool uses the national language set for you for displaying messages or menus. You can easily change this language by altering the setting of the environment variable `LANG`.



---

# 14 Appendix

## 14.1 Basic Reliant UNIX commands

**at** Execute a command in the background at a later time, specified on the command line.

The command line might look like this:

```
at 8:45am Jun 09
```

```
command1
```

```
command2
```

```
CTRL d
```

If you do not specify a date for *at*, the command is executed at the specified time within the next 24 hours.

**banner** Display a message consisting of up to ten characters in large letters on standard output.

**batch** Place commands in a queue and execute them when system load levels permit.

The command line for this command might look like this:

```
batch
```

```
command1
```

```
command2
```

```
CTRL d
```

The commands you specify for *batch* may also be shell scripts.

**cancel** Cancel print jobs. See also *lpr* in the *Commands* manual [5].

**cat** Display the contents of a specified file on the terminal. You can suspend output to the screen by pressing **CTRL** s. As soon as you want to resume output, press **CTRL** q. To terminate output to the screen and return to the shell, press **DEL**.

**cd** Change the current directory. If you do not specify a path, this command places you in your *HOME* directory, otherwise it places you in the directory you specify.

**cp** Copy a file to a new file; the original file is retained unchanged.

---

|                     |                                                                                                                                                                                                                                                                                                 |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>cut</code>    | Cut certain fields from each line of a file. This command allows you to extract columns from a table, for example.                                                                                                                                                                              |
| <code>date</code>   | Display the date and time.                                                                                                                                                                                                                                                                      |
| <code>diff</code>   | Compare two files. The <i>diff</i> command displays the lines that differ and tells you what you have to make the contents of the second file identical to those of the first.                                                                                                                  |
| <code>echo</code>   | Display input on the terminal.                                                                                                                                                                                                                                                                  |
| <code>ed</code>     | Edit a line using the line editor. If the specified file does not exist, the <i>ed</i> command creates it.                                                                                                                                                                                      |
| <code>grep</code>   | Search file(s) for a specific pattern and output the lines that contain that pattern. If you specify more than one file, <i>grep</i> also outputs the names of the files that contain the pattern.                                                                                              |
| <code>kill</code>   | Terminate a process. You specify the process ID ( <i>PID</i> ) of the process you want to terminate. You can find out the process ID using the <i>ps</i> command.                                                                                                                               |
| <code>lex</code>    | Generate programs to be used for simple text analysis.                                                                                                                                                                                                                                          |
| <code>lp</code>     | Output the contents of a specified file to a line printer.                                                                                                                                                                                                                                      |
| <code>lpr</code>    | Output the contents of a specified file to a line printer. <i>lpr</i> provides more options than <i>lp</i> for controlling print jobs. For example, it also allows you to request information on or cancel print jobs.                                                                          |
| <code>lpstat</code> | Display the status of all requests sent to the line printer. You are provided with special options for requesting certain information.                                                                                                                                                          |
| <code>ls</code>     | List the names of all files and directories except those that start with a dot (.). You are provided with special options for requesting more detailed information on files within directories. See also <i>ls</i> in the <i>Commands</i> manual [5].                                           |
| <code>mail</code>   | Display consecutively all messages sent to your terminal via the electronic mail facility. <i>mail</i> displays ? after each message to prompt you to take certain action, e.g. to save, forward or delete your message. You can enter another ? to obtain a list of all the available options. |

If *mail* is followed by a login name, the command sends a message to the owner of this login name. A message can have any length. Once you have finished typing in your message, enter **CTRL** d or a dot (.) to forward it to the recipient. You terminate a mail session by pressing **END**.

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                     |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>mailx</code> | <i>mailx</i> is easier to use and has a wider range of functions than <i>mail</i> .                                                                                                                                                                                                                                                                                                                                 |
| <code>mkdir</code> | Create a new directory. The new directory becomes a subdirectory of your current directory. You can create new files and directories in each directory.                                                                                                                                                                                                                                                             |
| <code>mv</code>    | Move a file to a new location within the file system. You can move a file to a different directory or rename it.                                                                                                                                                                                                                                                                                                    |
| <code>nohup</code> | Place a command in background mode so that it continues to execute even if you log out. The error messages and results are written to a file called <i>nohup.out</i> .                                                                                                                                                                                                                                              |
| <code>pg</code>    | Display the contents of a specified file one page at a time on your terminal. After displaying each page, <i>pg</i> waits for your instructions before proceeding.                                                                                                                                                                                                                                                  |
| <code>pr</code>    | Display a partially formatted version of a specified file on your terminal.                                                                                                                                                                                                                                                                                                                                         |
| <code>ps</code>    | Output the status and process ID of all processes that are currently executing. The <i>ps</i> command displays the status of all active jobs except those that are in the <i>at</i> or <i>batch</i> queue.                                                                                                                                                                                                          |
| <code>pwd</code>   | Display the absolute path name of the directory in which you are currently working.                                                                                                                                                                                                                                                                                                                                 |
| <code>rm</code>    | Delete a file from the file system. The <i>rm</i> command allows you to specify wildcard characters. You should use these with care, however, since it is not possible to restore a file once you have deleted it.                                                                                                                                                                                                  |
| <code>rmdir</code> | Delete a directory. The specified directory may not be the current directory. Since this command deletes only empty directories, you must delete any subdirectories and files contained in the directory you want to delete before entering this command. To delete a directory that is not empty, use the <i>rm</i> command with option <i>-r</i> . See also the relevant entry in the <i>Commands</i> manual [5]. |

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>sort</code>  | <p>Sort the lines of a file in a specified sequence and display the results on the screen.</p> <p>The ASCII table uses the following collating sequence:</p> <ol style="list-style-type: none"><li>1. Special characters</li><li>2. Numbers before letters</li><li>3. Uppercase before lowercase</li><li>4. Alphabetical order</li></ol> <p>There are several other options for sorting a file. All the <i>sort</i> options are listed under <i>sort</i> in the <i>Commands</i> manual [5].</p> |
| <code>spell</code> | <p>Check the spelling in a file. This command checks the spelling of the words in a specified file against an English dictionary. It lists any words that are not contained in the dictionary or cannot be derived from words in the dictionary (e.g. by adding a prefix or suffix).</p>                                                                                                                                                                                                        |
| <code>stty</code>  | <p>Display the configuration settings of specific I/O options for your terminal. Certain options and arguments of the <i>stty</i> command also allow you to reset these I/O options. See also the <i>stty</i> entry in the <i>Commands</i> manual [5].</p>                                                                                                                                                                                                                                      |
| <code>uname</code> | <p>Display the name of the Reliant UNIX system on which you are currently working.</p>                                                                                                                                                                                                                                                                                                                                                                                                          |
| <code>vi</code>    | <p>Edit a file using the display editor <i>vi</i>. If the file you specify does not exist, <i>vi</i> creates it. For detailed information on <i>vi</i>, see chapter 6 <i>The display editor vi</i>.</p>                                                                                                                                                                                                                                                                                         |
| <code>wc</code>    | <p>Count the number of lines, words and characters in the specified file and display the results on your terminal.</p>                                                                                                                                                                                                                                                                                                                                                                          |
| <code>who</code>   | <p>Display the login names of all users who are currently logged in on your Reliant UNIX system. For each login name, this command lists the terminal ID and the login time.</p>                                                                                                                                                                                                                                                                                                                |

## 14.2 Regular expressions

Regular expressions are used to scan a text for strings that match a defined pattern. Patterns are strings of characters, including numeric characters and metacharacters. Certain characters in the pattern may represent other characters, i.e. a regular expression stands for a set of strings.

A regular expression comprises a string of characters. These may be

- ordinary characters
- metacharacters

Ordinary characters are all the characters in the character set except the newline and metacharacters. Within a pattern, ordinary characters match themselves, e.g. the pattern *abc* will match only those strings that contain the character sequence *abc* anywhere in them.

Metacharacters, on the other hand, have special meanings when encountered in patterns. They are described in the *Metacharacters* table (see page 456).

There are two forms of regular expression:

- simple regular expressions
- extended regular expressions

The following sections describe the syntax of these forms of regular expression.

The following table shows which commands support regular expressions:

| Command        | Regular expression form |
|----------------|-------------------------|
| <i>awk</i>     | extended                |
| <i>bfs</i>     | simple                  |
| <i>csplit</i>  | simple                  |
| <i>ed</i>      | simple                  |
| <i>egrep</i>   | extended                |
| <i>ex</i>      | see note                |
| <i>expr</i>    | simple                  |
| <i>extract</i> | simple                  |
| <i>grep</i>    | simple                  |
| <i>lex</i>     | extended                |
| <i>nl</i>      | simple                  |
| <i>pg</i>      | simple                  |
| <i>sed</i>     | simple                  |
| <i>vi</i>      | see note                |

Table 128: Commands that support regular expressions

**Note:**

The *ex* and *vi* commands support regular expressions which differ slightly from the simple regular expressions described here. The special points that apply to *ex* and *vi* are described under *ed* and under *vi* in the *Commands* manual [5].



Simple regular expressions

Simple regular expressions are constructed as follows:

| No. | Regular expression | Stands for                                                                                                                                                                                                                                                                                                                                                         | Example                                         | Matching strings                                 |
|-----|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------|--------------------------------------------------|
| 1   | c                  | The character <i>c</i> , where <i>c</i> is not a metacharacter.                                                                                                                                                                                                                                                                                                    | a                                               | a                                                |
| 2   | \ c                | The character <i>c</i> , where <i>c</i> can be any character other than<br>( ) { } 1 2 3 4 5 6 7 8 9<br>Regular expressions in this form are meaningful if <i>c</i> is a metacharacter. \ <i>c</i> then stands for character <i>c</i> itself, as the backslash escapes its special meaning as a metacharacter.                                                     | \ a<br><br>\ *                                  | a<br><br>*                                       |
| 3   | .                  | Any single character.                                                                                                                                                                                                                                                                                                                                              | .                                               | a, x, *, ...                                     |
| 4   | [s]                | Any character from <i>s</i> , where <i>s</i> is a set of characters. If the right square bracket ] is to be in this set, it has to be placed first in the set. If a hyphen is to be one of the characters in the set, it has to be placed first or last. If a caret / circumflex ^ is to be one of the characters in the set, it can be placed anywhere but first. | [mz]<br><br>[a]<br><br>[-a]<br>[a-]<br><br>[a^] | m, z<br><br>], a<br><br>-, a<br>-, a<br><br>a, ^ |

Table 129: Simple regular expressions

| No. | Regular expression   | Stands for                                                                                                                                                                                                                                                                                                                                                         | Example                                    | Matching strings                                                                                                                                                               |
|-----|----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|     | [c1-c2]              | <p>Any character in the range <i>c1</i> through <i>c2</i> inclusive in the ASCII collating sequence.</p> <p><i>c1</i> must come before <i>c2</i> in the ASCII collating sequence. If it does not, <i>c1-c2</i> does not denote a range but simply stands for the two characters <i>c1</i> and <i>c2</i>.</p> <p>The two forms can be combined:<br/>[s1c1-c2s2]</p> | <p>[a-m]</p> <p>[m-a]</p> <p>[ado-qxz]</p> | <p>a, m and any character in between in the ASCII collating sequence</p> <p>m, a</p> <p>a, d, o, q, x, z and any character between o and q in the ASCII collating sequence</p> |
| 5   | [^s]<br><br>[^c1-c2] | <p>Any character not included in set <i>s</i>.</p> <p>Any character not in the range <i>c1</i> through <i>c2</i> inclusive. Refer also to [<i>c1-c2</i>].</p>                                                                                                                                                                                                      | <p>[^xyz]</p> <p>[^0-9]</p>                | <p>Any character except x, y, z</p> <p>Any character except 0, 9 and all characters between 0 and 9 in the ASCII collating sequence</p>                                        |

Table 129: Simple regular expressions

| No. | Regular expression | Stands for                                                                                                                                                                    | Example  | Matching strings                                                                                   |
|-----|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------|----------------------------------------------------------------------------------------------------|
|     |                    | The two forms can be combined:<br>[^s1c1-c2s2]                                                                                                                                | [^a0-9b] | Any character except a, b, 0, 9 and all characters between 0 and 9 in the ASCII collating sequence |
| 6   | <b>r*</b>          | Zero, one or more occurrences of regular expression <i>r</i> .<br><i>r</i> has to be of form 1 - 5, 12, 15 or 16.                                                             | a*       | Nothing, a, aa, aaa, ...                                                                           |
| 7   | <b>r\{m,n\}</b>    | At least <i>m</i> and at most <i>n</i> occurrences of regular expression <i>r</i> .<br><i>r</i> must be of form 1 - 5, 12, 15 or 16.                                          | a\{1,2\} | a or aa                                                                                            |
|     | <b>r\{m\}</b>      | Precisely <i>m</i> occurrences of regular expression <i>r</i> .<br><i>r</i> must be of form 1 - 5, 12, 15 or 16.                                                              | a\{3\}   | aaa                                                                                                |
|     | <b>r\{m,\}</b>     | At least <i>m</i> occurrences of regular expression <i>r</i> .<br><i>r</i> must be of form 1 - 5, 12, 15 or 16.                                                               | a\{3,\}  | aaa, aaaa, aaaaa, ...                                                                              |
| 8   | <b>rx</b>          | (Concatenation) An occurrence of regular expression <i>r</i> followed by an occurrence of regular expression <i>x</i> . <i>r</i> and <i>x</i> can be any regular expressions. | [ab].    | ax, a3, a*, bz, ...                                                                                |

Table 129: Simple regular expressions

| No. | Regular expression   | Stands for                                                                                                                                                                                                                                                                                                                                  | Example                                                                                        | Matching strings                                       |
|-----|----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------|--------------------------------------------------------|
| 9   | <code>^r</code>      | An occurrence of regular expression <i>r</i> appearing at the start of a line, i.e. straight after a newline character or at the start of a line.<br><i>r</i> can be a regular expression in any form other than number 9.                                                                                                                  | <code>^[aA]pple</code>                                                                         | apple or Apple at the start of a line                  |
| 10  | <code>r\$</code>     | An occurrence of regular expression <i>r</i> at the end of a line, i.e. directly before a newline character.<br><i>r</i> can be a regular expression in any form other than number 10.                                                                                                                                                      | <code>[bB]arge\$</code>                                                                        | barge or Barge at the end of a line                    |
| 11  | <code>\ (r\ )</code> | Occurrences of regular expression <i>r</i> .<br><i>r</i> can be any regular expression. Only useful together with No. 12.                                                                                                                                                                                                                   | <code>\ ([aA]pple\ )</code>                                                                    | apple, Apple                                           |
| 12  | <code>\ n</code>     | <i>n</i> is an integer in the range 1 through 9. <code>\ n</code> appearing in a concatenated regular expression stands for the regular expression <i>x</i> , where <i>x</i> is the <i>n</i> th regular expression enclosed in <code>\</code> (and <code>\</code> ) sequences that appeared earlier in the concatenated regular expression. | <code>\ (a\ (b\ )\ )\ 2</code><br><code>\ (ha\l\ )li\ 1lo</code><br><code>\ (ab\ )x\ 1*</code> | abb<br>hallihallo<br>abx,<br>abxab,<br>abxabab,<br>... |

Table 129: Simple regular expressions

Metacharacters

| Metacharacter                                                                   | The character to the left has a special meaning if                                                                                                                   |
|---------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| \                                                                               | - it is not preceded by a backslash \                                                                                                                                |
| .<br>[                                                                          | - it is not preceded by a backslash \ and<br>- it does not appear between [ and ]                                                                                    |
| *                                                                               | - it is not preceded by a backslash \ ,<br>- it does not appear between [ and ],<br>- it is not the first character in a pattern and<br>- it does not come after \ ) |
| \$                                                                              | - it is the last character in a pattern                                                                                                                              |
| ^                                                                               | - it is the first character in a pattern or<br>- it is the first character in square brackets [...]                                                                  |
| -                                                                               | - it is in square brackets but not placed first or last                                                                                                              |
| Regular<br>expression<br>delimiter<br>such as<br>/.../ or<br>?...? in <i>ed</i> | - it is not preceded by a backslash \                                                                                                                                |

Table 130: Metacharacters in regular expressions

Extended regular expressions

Extended regular expressions include full regular expression syntax, with the following exception:  
The `\(...\)` construct used in simple regular expressions does *not* have a special meaning in extended regular expression syntax; for example, the extended regular expression `\(ab\)` stands for the character string `(ab)`.  
Extended regular expressions provide the following additional syntax elements for the construction of patterns:

| No. | Regular expression   | Stands for                                                                                            | Example                                          | Matching strings                                                               |
|-----|----------------------|-------------------------------------------------------------------------------------------------------|--------------------------------------------------|--------------------------------------------------------------------------------|
| 13  | <code>r+</code>      | One or more occurrences of regular expression <i>r</i> . <i>r</i> must be of form 1 - 5, 15 or 16.    | <code>u+</code>                                  | <code>u</code> , <code>uu</code> , <code>uuu</code> , ...                      |
| 14  | <code>r?</code>      | Zero or one occurrence of regular expression <i>r</i> . <i>r</i> must be of the form 1 - 5, 15 or 16. | <code>u?</code>                                  | nothing or <code>u</code>                                                      |
| 15  | <code>(r)</code>     | Strings matching regular expression <i>r</i> . <i>r</i> can be any regular expression.                | <code>(ok(abc))</code><br><br><code>(au)*</code> | <code>okabc</code><br><br>nothing or <code>au</code> , <code>auau</code> , ... |
| 16  | <code>(r1?r2)</code> | Strings matching regular expression <i>r1</i> or regular expression <i>r2</i> .                       | <code>(ok?ko)</code>                             | <code>ok</code> or <code>ko</code>                                             |

Table 131: Extended regular expressions

Precedence

The following table shows the precedence of operators in regular expressions.

| Operator        | Precedence         |
|-----------------|--------------------|
| [...] (...)     | Highest precedence |
| * ? + \ {...\ } | .                  |
| Concatenation   | .                  |
|                 | Lowest precedence  |

Table 132: Precedence of operators in regular expressions

Examples

1. Simple regular expressions

| Pattern  | Meaning                                            | Matching strings       |
|----------|----------------------------------------------------|------------------------|
| ab.d     | a - b - any one character - d                      | abcd, abXd, ab*d, ...  |
| ab.*d    | a - b - any string (including the null string) - d | abd, abxd, abX*Yd, ... |
| ab[xyz]d | a - b - x or y or z - d                            | abxd, abyd, abzd       |
| ab[^c]d  | a - b - any character other than c - d             | abbd, abXd, ab*d, ...  |
| ^abcd\$  | a line containing only the string abcd             |                        |

Table 133: Examples of simple regular expressions

2. Extended regular expressions

| Pattern   | Meaning                                            | Matching strings  |
|-----------|----------------------------------------------------|-------------------|
| ab.+d     | a - b - any sequence of one or more characters - d | abjd, abX*Yd, ... |
| abc?d     | a - b - c or nothing - d                           | abd, abcd         |
| (abc xyz) | abc or xyz                                         | abc, xyz          |

Table 134: Examples of extended regular expressions

# 14.3 ISO 646 character set

The following table combines the international version of the ISO 646 character set, the US version (equivalent to the 7-bit US ASCII coded character set), the British version and the German version.

| decimal | octal | hexa-dec. |     | Meaning                                                                                    | Control |
|---------|-------|-----------|-----|--------------------------------------------------------------------------------------------|---------|
| 0       | 00    | 00        | NUL | Null, no operation                                                                         | @       |
| 1       | 01    | 01        | SOH | Start of Heading                                                                           | A       |
| 2       | 02    | 02        | STX | Start of Text                                                                              | B       |
| 3       | 03    | 03        | ETX | End of Text                                                                                | C       |
| 4       | 04    | 04        | EOT | End of Transmission, key <span style="border: 1px solid black; padding: 0 2px;">END</span> | D       |
| 5       | 05    | 05        | ENQ | Enquiry                                                                                    | E       |
| 6       | 06    | 06        | ACK | Acknowledge                                                                                | F       |
| 7       | 07    | 07        | BEL | Bell                                                                                       | G       |
| 8       | 10    | 08        | BS  | Backspace                                                                                  | H       |
| 9       | 11    | 09        | HT  | Horizontal Tabulation                                                                      | I       |
| 10      | 12    | 0A        | LF  | Line Feed                                                                                  | J       |
| 11      | 13    | 0B        | VT  | Vertical Tabulation                                                                        | K       |
| 12      | 14    | 0C        | FF  | Form Feed                                                                                  | L       |
| 13      | 15    | 0D        | CR  | Carriage Return                                                                            | M       |
| 14      | 16    | 0E        | SO  | Shift Out                                                                                  | N       |
| 15      | 17    | 0F        | SI  | Shift In                                                                                   | O       |
| 16      | 20    | 10        | DLE | Data Link Escape                                                                           | P       |
| 17      | 21    | 11        | DC1 | Device Control 1 (turn output on)                                                          | Q       |
| 18      | 22    | 12        | DC2 | Device Control 2                                                                           | R       |
| 19      | 23    | 13        | DC3 | Device Control 3 (turn output off)                                                         | S       |
| 20      | 24    | 14        | DC4 | Device Control 4                                                                           | T       |
| 21      | 25    | 15        | NAK | Negative Acknowledge                                                                       | U       |
| 22      | 26    | 16        | SYN | Synchronous Idle                                                                           | V       |
| 23      | 27    | 17        | ETB | End of Transmission Block                                                                  | W       |

Table 135: ISO 646 character set



| decim-al | octal | hexa-dec. |     | Meaning              | Control |
|----------|-------|-----------|-----|----------------------|---------|
| 24       | 30    | 18        | CAN | Cancel               | X       |
| 25       | 31    | 19        | EM  | End of Medium        | \ or    |
| 26       | 32    | 1A        | SUB | Substitute Character | Z       |
| 27       | 33    | 1B        | ESC | Escape               |         |
| 28       | 34    | 1C        | FS  | File Separator       | \       |
| 29       | 35    | 1D        | GS  | Group Separator      | ]       |
| 30       | 36    | 1E        | RS  | Record Separator     | ^       |
| 32       | 40    | 20        | SP  | SPACE                |         |
| 33       | 41    | 21        | !   |                      |         |
| 34       | 42    | 22        | "   |                      |         |
| 35       | 43    | 23        | #   | see <sup>1</sup>     |         |
| 36       | 44    | 24        | \$  | see <sup>2</sup>     |         |
| 37       | 45    | 25        | %   |                      |         |
| 38       | 46    | 26        | &   |                      |         |
| 39       | 47    | 27        | '   |                      |         |
| 40       | 50    | 28        | (   |                      |         |
| 41       | 51    | 29        | )   |                      |         |
| 42       | 52    | 2A        | *   |                      |         |
| 43       | 53    | 2B        | +   |                      |         |
| 44       | 54    | 2C        | ,   |                      |         |
| 45       | 55    | 2D        | -   |                      |         |
| 46       | 56    | 2E        | .   |                      |         |
| 47       | 57    | 2F        | /   |                      |         |
| 48       | 60    | 30        | 0   |                      |         |
| 49       | 61    | 31        | 1   |                      |         |
| 50       | 62    | 32        | 2   |                      |         |
| 51       | 63    | 33        | 3   |                      |         |
| 52       | 64    | 34        | 4   |                      |         |

Table 135: ISO 646 character set

| decim-<br>al | octal | hexa-<br>dec. |   | Meaning | Control |
|--------------|-------|---------------|---|---------|---------|
| 53           | 65    | 35            | 5 |         |         |
| 54           | 66    | 36            | 6 |         |         |
| 55           | 67    | 37            | 7 |         |         |
| 56           | 70    | 38            | 8 |         |         |
| 57           | 71    | 39            | 9 |         |         |
| 58           | 72    | 3A            | : |         |         |
| 59           | 73    | 3B            | ; |         |         |
| 60           | 74    | 3C            | < |         |         |
| 61           | 75    | 3D            | = |         |         |
| 62           | 76    | 3E            | > |         |         |
| 63           | 77    | 3F            | ? |         |         |
| 64           | 100   | 40            | @ |         |         |
| 65           | 101   | 41            | A |         |         |
| 66           | 102   | 42            | B |         |         |
| 67           | 103   | 43            | C |         |         |
| 68           | 104   | 44            | D |         |         |
| 69           | 105   | 45            | E |         |         |
| 70           | 106   | 46            | F |         |         |
| 71           | 107   | 47            | G |         |         |
| 72           | 110   | 48            | H |         |         |
| 73           | 111   | 49            | I |         |         |
| 74           | 112   | 4A            | J |         |         |
| 75           | 113   | 4B            | K |         |         |
| 76           | 114   | 4C            | L |         |         |
| 77           | 115   | 4D            | M |         |         |
| 78           | 116   | 4E            | N |         |         |
| 79           | 117   | 4F            | O |         |         |
| 80           | 120   | 50            | P |         |         |

Table 135: ISO 646 character set

| decim-<br>al | octal | hexa-<br>dec. |   | Meaning                      | Control |
|--------------|-------|---------------|---|------------------------------|---------|
| 81           | 121   | 51            | Q |                              |         |
| 82           | 122   | 52            | R |                              |         |
| 83           | 123   | 53            | S |                              |         |
| 84           | 124   | 54            | T |                              |         |
| 85           | 125   | 55            | U |                              |         |
| 86           | 126   | 56            | V |                              |         |
| 87           | 127   | 57            | W |                              |         |
| 88           | 130   | 58            | X |                              |         |
| 89           | 131   | 59            | Y |                              |         |
| 90           | 132   | 5A            | Z |                              |         |
| 91           | 133   | 5B            | [ | see <sup>3</sup> ; German: Ä |         |
| 92           | 134   | 5C            | \ | see <sup>1</sup> ; German: Ö |         |
| 93           | 135   | 5D            | ] | see <sup>1</sup> ; German: Ü |         |
| 94           | 136   | 5E            | ^ |                              |         |
| 95           | 137   | 5F            | _ |                              |         |
| 96           | 140   | 60            | ` |                              |         |
| 97           | 141   | 61            | a |                              |         |
| 98           | 142   | 62            | b |                              |         |
| 99           | 143   | 63            | c |                              |         |
| 100          | 144   | 64            | d |                              |         |
| 101          | 145   | 65            | e |                              |         |
| 102          | 146   | 66            | f |                              |         |
| 103          | 147   | 67            | g |                              |         |
| 104          | 150   | 68            | h |                              |         |
| 105          | 151   | 69            | i |                              |         |
| 106          | 152   | 6A            | j |                              |         |
| 107          | 153   | 6B            | k |                              |         |
| 108          | 154   | 6C            | l |                              |         |

Table 135: ISO 646 character set

| decimal | octal | hexa-dec. |     | Meaning                      | Control |
|---------|-------|-----------|-----|------------------------------|---------|
| 109     | 155   | 6D        | m   |                              |         |
| 110     | 156   | 6E        | n   |                              |         |
| 111     | 157   | 6F        | o   |                              |         |
| 112     | 160   | 70        | p   |                              |         |
| 113     | 161   | 71        | q   |                              |         |
| 114     | 162   | 72        | r   |                              |         |
| 115     | 163   | 73        | s   |                              |         |
| 116     | 164   | 74        | t   |                              |         |
| 117     | 165   | 75        | u   |                              |         |
| 118     | 166   | 76        | v   |                              |         |
| 119     | 167   | 77        | w   |                              |         |
| 120     | 170   | 78        | x   |                              |         |
| 121     | 171   | 79        | y   |                              |         |
| 122     | 172   | 7A        | z   |                              |         |
| 123     | 173   | 7B        | {   | see <sup>4</sup> ; German: ä |         |
| 124     | 174   | 7C        |     | see <sup>1</sup> ; German: ö |         |
| 125     | 175   | 7D        |     | see <sup>1</sup> ; German: ü |         |
| 126     | 176   | 7E        |     | see <sup>5</sup> ; German: ß |         |
| 127     | 177   | 7F        | DEL | Delete                       |         |

Table 135: ISO 646 character set

- <sup>1</sup>ASCII: as international version (see table)  
British: as international version (see table)  
German: umlaut forms (see table)
- <sup>2</sup>ASCII: as international version (see table)  
British: as international version (see table)  
German: umlaut forms (see table)
- <sup>3</sup>ASCII: as international version (see table)  
British: as international version (see table)  
German: umlaut forms (see table)
- <sup>4</sup>ASCII: as international version (see table)  
British: as international version (see table)  
German: umlaut forms (see table)

<sup>5</sup>ASCII: ~

British: ~

German: ß



---

# Glossary

## absolute path name

A path name that originates at the root directory of the Reliant UNIX file system and leads to a specific file or directory. Each file and directory in the Reliant UNIX system has a unique absolute path name. See *path name*, *relative path name*.

## address

Used to access files or specific parts of a program. An address may be a word that identifies a storage location, a contiguous storage area or a functional unit, or part of an editor command that specifies a line number or range.

## append mode

A text editing mode in which the characters you type are entered as text into the text editor's buffer. In this mode you enter text to the right of the current cursor position in the buffer. See *input mode*, *command mode* and *insert mode*.

## archive

Archives are also known as archive files or backup files. An archive is a special file in which you back up several files or directories in one access onto floppy disk, tape cartridge or magnetic tape. You use the *tar* or *cpio* command to create an archive. An archive can contain several files and directories and is terminated automatically with an end-of-archive mark.

## argument

In general, an argument is the variable part of a function. Specifically, it is the element of the command line that specifies the data on which a command is to operate. Arguments follow the command name and can include numbers, letters or strings. For example, in the command line *lp -m my\_file*, *lp* is the command and *my\_file* is the argument. See *option*.

## ASCII

The acronym for **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange, used by Reliant UNIX. This standard 7-bit code for data transmission represents 128 characters, including letters, numbers and standard special characters such as #, \$, % and &. In addition, Reliant UNIX often uses the extended 8-bit ASCII code, which represents 256 characters.

### backend

An interface program for printer output. Each printer type has its own backend. Backends ensure that the device interface is set correctly and data is formatted and output correctly to the corresponding printer.

### background

A type of program execution that does not involve interaction between the user and the program. While the program is running, the shell prompt is displayed and other commands can be issued at the terminal. See *foreground*.

### batch processing

Operating system mode in which large quantities of data are processed according to a predefined schedule. Data required for a particular program is collected and processed in a single pass. The next program is then executed. See *interactive data processing*.

### baud rate

Also known as the modulation rate, the baud rate is a measure of the speed of data transfer between a computer and a peripheral device (e.g. a terminal) or between two devices. 1 baud = 1 signal element per second; if purely binary characters are being transferred, 1 baud = 1 bit/second. Some common baud rates in the telephone network are 2400, 9600 and 19200 bits/second. Computer-internal data transfer speeds may reach 10 Megabits/second.

### block

See *device*.

### buffer

A storage area used wherever data is exchanged or transferred at different speeds within a system. Buffering adjusts the timing and reduces the load on the faster unit involved.

### CD-ROM

Abbreviation for **C**ompact **D**isc **R**ead **O**nly **M**emory. A CD-ROM is an off-line optical storage medium consisting of a circular plastic disk to which digital characters are written once only using lasers. The data is read without contact by means of a laser beam. CD-ROMs are normally 5 1/4" in diameter and have a storage capacity of up to 650 Mbytes.



**character**

See *device*.

**command**

An instruction in a command language that requests the system to perform a particular operation or execute a particular program. A command is often the name of a file that contains a program.

**command execution**

The commands that appear one after the other in a program are executed consecutively without interruption.

**command file**

See *executable file*, *shell script*.

**command interpreter**

See *shell*.

**command line**

A line containing one or more commands, terminated by pressing `[Enter]`. Command lines instruct the computer to perform one or more tasks. They can also contain options and arguments for the commands.

**command mode**

A text editing mode in which the characters you enter are interpreted as editing commands. This mode allows you to move around the buffer, delete text and move lines of text. See *input mode*, *append mode* and *insert mode*.

**control character**

Control characters are complete bytes, words or signals that the system uses to control individual functions or operations. Control characters are nonprinting characters that you enter by typing a character while holding down the `[CTRL]` key, for example. A synonym is *escape sequence*, so called because the control character is generated using `[ESC]`.

**current directory**

The directory in which you are currently working. You have direct access to all files and subdirectories contained in your current directory. The shorthand notation for the current directory is the dot (`.`).

### cursor

A cue displayed on the screen that indicates the position at which you can enter or delete a character. It is usually a rectangle or a blinking underscore character.

### cylinder

Cylinders are tracks in the same location on opposite sides of a floppy disk or hard disk that can be accessed simultaneously by two read/write heads.

### default

An automatically assigned value or condition that is valid unless you explicitly change it.

### delimiter

A metacharacter that logically separates certain data, e.g. the words or arguments in a command line. The most frequently used delimiters under Reliant UNIX are the blank and the tab.

### device

A general term for units of hardware. The most important devices in a data processing system are the CPU, the *peripheral devices*, the *input/output* devices and storage devices, and *teleprocessing* equipment. A distinction is made between character devices (such as terminals), which store or transmit characters one at a time, and block devices (such as hard disks), which store or transmit data in blocks of characters, e.g. 512 or 1024.

### device driver

A software module that is part of the kernel. It supports access to a peripheral device by converting the corresponding control impulses and passing them to the relevant device.

### device number

A number that describes exactly one block or character device. A device number consists of:  
the major device number, which identifies the device type  
the minor device number, which identifies a subunit of a device (e.g. a partition on a hard disk)

**directory**

Also known as catalogs, directories are a type of file used to group and organize other files in hierarchical form. Directories contain a list of subdirectories and files. You can use certain commands to view the contents of a directory and obtain information on the files contained in it. For further information, see chapter 3 *The file system*.

**DOS**

Abbreviation for **Disk Operating System**. This is the name of a widely used operating system for personal computers. Developed and marketed by Microsoft under the name MS-DOS, it is also used by IBM under the name PC-DOS and has been developed into DR-DOS by Digital Research/Novell.

**duplex**

Also known as full-duplex. A method of data transmission that allows a computer system to send and receive data at the same time. Terminals and modems normally work in half-duplex mode; data transmission in the Reliant UNIX system is full-duplex.

**electronic mail**

The feature of an operating system that allows computer users to exchange messages via the computer. Reliant UNIX provides you with the two commands *mail* and *mailx* for this purpose.

**environment**

The conditions that apply to one or more processes when you use the Reliant UNIX system. Your environment includes settings that personalize your login and allow you to interact in specific ways with the Reliant UNIX system and the computer. For example, your shell environment includes the setting for your shell prompt string, backspace and erase characters, and commands for sending output from your terminal to the computer.

**erase character**

The character you type to delete the character you typed previously. The default Reliant UNIX erase character is .

**error message**

A message displayed on your terminal to inform you that an error has occurred during command or program execution.

### executable file

A file containing machine code or commands that can be executed by the computer. It may be a compiled program or a shell script, for example. When you enter the file name on the command line, the commands contained in the file are executed. See *command execution*, *shell script*, *source code*.

### file

A collection of information in the form of a sequence of characters. A file is used to store data required for particular purposes and tasks in an ordered, safe form so that you can access it as required. A file may contain data, programs or other text. See *ordinary file*, *executable file*, *directory*, *pipe*.

### file name

A series of characters that describes a file. Under Reliant UNIX, file names may not contain slashes (/).

### file system

A collection of files and the structure that links them together. In the Reliant UNIX file system, the directories and the files contained in them are organized in a hierarchical structure. For further information, see chapter 3 *The file system*.

### filter

A command that reads from standard input, processes the input and writes the result to standard output.

### floppy disk

A common off-line storage medium consisting of a circular disk of flexible plastic with a magnetic coating, enclosed in a protective envelope. It remains in this envelope even when you are using it. Several sizes of floppy disk were developed consecutively. Common floppy disk types are those with a diameter of 5 1/4" in a flexible envelope and those with a diameter of 3 1/2" in a rigid plastic casing. See also *cylinder*, *track*, *label area*.

### foreground

When executing a command in the foreground, the shell waits for one command to terminate before prompting you for another. In other words, if you pass an instruction to the computer, you must wait for the computer to respond before entering the next command. See also *background*.

**full-screen editor**

A text editing program that displays the text file or a section of the text you want to edit on the screen. You can add and modify text at any point in the file that is visible on the screen. To do this, you first move the cursor to the appropriate position. The changes you make have an immediate effect on the screen display. See *text editor*, *line editor*.

**hard disk**

An important off-line storage medium that can hold large quantities of data for fast direct access. It consists basically of a circular ceramic or metal disk coated in a magnetic substance. Both its surfaces are formatted with concentric tracks for bit-serial data storage. Hard disks today are usually 3 1/2" or 5 1/4" in diameter.

**hardware**

The sum of the physical components of a computer and any associated peripheral devices.

**hidden character**

One of a group of characters within the ASCII character set that cannot be represented on the screen, such as `␣`, the escape character and the key combination `CTRL d`.

**home directory**

This directory automatically becomes your current directory when you enter the `cd` command on its own. The path name of the home directory is defined in the `HOME` variable. You can view the contents of the `HOME` variable using the `echo $HOME` command. Provided you do not change the contents of the `HOME` variable, your home directory is also your login directory.

**input/output**

Often abbreviated to I/O, this is the path by which information enters a computer and leaves the system. For example, an input device is the terminal keyboard and an output device is the terminal screen.

**input mode**

A text editing mode in which the characters you type in are entered as text in the text editor's buffer. If you want to issue a command, you must exit input mode. See *command mode*, *append mode* and *insert mode*.

### **insert mode**

A text editing mode in which the characters you type in are entered as text into the text editor's buffer. In this mode you enter text before the current position in the buffer. See *input mode*, *append mode* and *command mode*.

### **interactive data processing**

Interactive data processing on computers is characterized by continual human intervention, e.g. data entry. A dialog is conducted directly between the user and the computer.

### **keyboard buffer**

The buffer into which Reliant UNIX reads your input while sending you information in response to previous input. See *buffer*.

### **label area**

The label area of a floppy disk is the area in which the owner of the floppy disk and the version number of its contents can be entered. The label area is always located on cylinder 0. Once you have formatted a floppy disk with the corresponding special file, no more data can be written to the label area until you reformat the floppy disk.

### **line editor**

An editing program that processes the contents of a file on a line-by-line basis. You use line addresses to specify the order in which you want to edit the individual lines. The changes you make are immediately visible on the screen. See *text editor*, *screen editor*.

### **local system**

A local system is a system in which data transfer between peripheral devices and the CPU takes place only via the channels of the CPU, not via teleprocessing equipment. See *remote system*.

### **log out**

The procedure used to exit from the Reliant UNIX operating system.

### **login**

To use a terminal or a remote computer, you must first log in to Reliant UNIX. The system's welcome screen appears on your terminal, prompting you to enter your login name and, if any, your password.

See *login name*, *password*.

**login directory**

The directory in which you are located automatically when you log in. See *home directory*.

**login name**

A series of alphanumeric characters that identify a user. Each user is assigned a unique login name.

**magnetic tape**

The oldest type of off-line magnetic storage. Magnetic tapes commonly used today are 1/2" (12.7 mm) wide and approximately 0.03 mm thick. Common tape lengths are 2400 ft. (731.5 m), 1200 ft. (365.7 m) and 600 ft. (182.8 m). Data is stored physically in the order in which it is written to the tape. To protect it from external interference, the tape is stored rolled up on a reel.

**mode**

In general, the way in which data is processed in a computer or transferred. Specifically, it is a particular type of program operation, e.g. an editor's append mode.

**modem**

Abbreviation for **modulator/demodulator**. A device that enables a terminal and a computer to communicate via a telephone line. A modem converts digital signals into tones and vice versa, permitting data exchange across the public telephone network.

**multitasking**

A multitasking operating system contains several independent I/O controllers that can be initiated by the program controller. This type of operating system appears to execute several programs simultaneously.

**multiuser**

A multiuser operating system allows several users to work with a computer at the same time, each using their own peripheral device.

**network**

A complex structure of lines and control equipment that allows the stations attached to the network to exchange data. There are various different types of network architecture, characterized by the type and configuration of the lines and the way in which the network is controlled.

### numbering system

A system used to represent quantitative attributes of objects or procedures. The numbers are formed from digits. The values of successive digit places are successive integral powers of a single radix, each multiplied by the same factor (radix numbering system).

A radix numbering system with base 2 is a binary numbering system, a system with base 10 is a decimal system, a system with base 8 is an octal system and a system with base 16 is a hexadecimal system.

### operating system

Collective term for the system software required to run a computer. You must have an operating system and hardware in order to use application programs. The operating system is stored on a random access storage unit. Some basic programs reside in the computer's main memory, while others are stored externally and loaded into main memory as required.

The function of an operating system is to:  
ensure that the computer and application programs run properly  
control the I/O system  
provide various utilities

### option

A special instruction that modifies how a command runs. An option is a type of argument that follows the command name and usually precedes other arguments on the command line. By convention, an option is preceded by a minus sign (-) to distinguish it from other arguments. See *argument*.

### ordinary file

A file that contains text, other data or an executable program. Pipes and directories are not ordinary files. See *executable file*.

### output

Information processed by the computer and delivered to you by way of a printer, terminal or a similar device.

### parameter

An initially undefined variable that is used to vary the context of a function, statement or command. You can vary the effect of a command by substituting actual values for a variable. See *positional parameter*.



**parent directory**

The directory immediately above a subdirectory or file in the file system. The shorthand notation for the parent directory is two dots (..).

**parity**

Parity is the checksum of the binary digits in a byte or word. The sum of the data bits and the parity bit must be either an even or odd number of binary digits to ensure data consistency. This method is used to check whether the data received matches the data sent.

**password**

A code word known only to you that you must specify when you log in. The computer uses the password to verify that you are authorized to access the system.

**path name**

Under Reliant UNIX, each file and each directory has a unique path name. The path name indicates the location of the file or directory within the file system and how to access it. The path name is a sequence of directory names separated by slashes (/). A slash at the start of a path name indicates an absolute path name. The path name ends with the name of a file or directory. There are two types of path names under Reliant UNIX: absolute and relative. For further information, see chapter 3 *The file system*.

**pattern**

A set of elementary signals or elements (e.g. bits or characters) arranged according to certain rules. Patterns are used to represent objects. See *regular expression*.

**peripheral device**

A device controlled by the computer, used mainly for input, output and storage functions. Peripheral devices include terminals, printers and disk drives.

**permissions**

Permissions define whether processes are authorized to access system objects, e.g. to write data to a file. Specifically, they are settings that grant or deny a user permission to read, modify or execute a particular directory or file. You can modify the permissions for your directories and files using the *chmod* command, and set defaults for them using *umask*.

### personal computer

Personal computers (PCs) are microcomputers used primarily by individuals at a specific workplace, in the office or at home. A distinction is made in terms of performance between home computers and professional workstations.

### pipe

A method of interprocess communication that supports data flow in one direction between dependent processes. It redirects the output of one command so that the next command uses it as input. The symbol for a pipe is the vertical line (`|`) on the shell command line.

### pipeline

A series of processes, or filters, in which the standard output of one process is redirected as input to the next process using the *pipe* symbol (see *pipe*). The last filter in the pipeline writes its results to standard output. You can redirect output into a file. See *filter*.

### positional parameter

A numbered variable used in a shell script to access the strings you specified as arguments on the command line when invoking the shell script. The name of the shell script itself, for example, is positional parameter `$0`. See *variable* and *shell script*.

### printer

An output device that prints the data it receives from the computer onto paper. Commonly used printers today include laser, needle and ink-jet printers.

### process

Execution of a series of actions by a computer. These actions form a functional unit with regard to a particular purpose. A process is characterized by the program that executes and the data it uses. A program may be assigned to several processes. On single-processor systems, processes run on a time-sharing basis; on multiprocessor systems, all processes execute simultaneously.

**program**

A program contains the instructions given to a computer in the form of commands. These commands constitute a syntactical unit in terms of the programming language used. A program is a finite algorithm that describes a task uniquely and in a form that the computer can understand. Programs are compiled into machine language according to the programming languages used. See *software*.

**prompt**

A cue displayed on your terminal by the shell to indicate that the system is ready to accept your input. The default prompt under Reliant UNIX is the dollar sign (\$).

**pseudo terminal**

Also known as a virtual terminal, this is a terminal that is defined only by its device type within a program. The program emulates a terminal that you use, for example, if you are working on a remote system and connected indirectly to a computer across a network.

**read/write head**

Technical equipment for reading and writing magnetic data media such as floppy disks, tape cartridges, etc.

**regular expression**

A regular expression is a string that specifies patterns and defines what patterns are covered by the string. Regular expressions are used to find occurrences of a predefined pattern in a text. Regular expressions may also include metacharacters that stand for other characters.

For example, the command `grep 'M[ea][iy]er' name_list` finds all the lines in a file called `name_list` that contain one of the following strings: Meier, Maier, Meyer or Mayer.

**relative path name**

The path to a file or directory in relation to the position of the current directory within the file system. See *path name*, *absolute path name*.

### Reliant UNIX

Reliant UNIX is a multiuser, multitasking operating system for use on personal computers. It consists of an operating system kernel for program control, memory management and file system management, and a processor, known as the shell, that controls the wide range of commands and acts as an interface between the user and all the system programs. About 90% of the operating system is written in the C programming language. Reliant UNIX V5.4x is a derivative of AT&T's UNIX System 5, Release 4 and meets the requirements of the X/Open Portability Guide.

### remote

Not located on your system. See *remote system*.

### remote system

A remote system is a system in which data is exchanged between users, peripheral devices and the CPU by means of teleprocessing equipment. See *local system*.

### root directory

The source directory of all files and directories in the file system. The root directory is represented by a slash (/).

### search pattern

See *string*.

### sector

A sector is a segment of all the tracks (circles) on a floppy disk. Sectors divide each track into an equal number of areas, known as blocks. The number of sectors on a floppy disk varies according to the format of the floppy disk.

### shell

The shell is also known as the command interpreter. It is a type of program architecture in which a program forms a shell that links you directly with a greater set of programs. This program is a direct interface between you and the system. You come into contact only with the shell and its commands; the shell handles all other programs. Three of the shells available are described under *sh*, *csh* and *ksh* in the *Commands* manual [5].

**shell script**

An executable file that is not a compiled program. It contains commands that are read and executed by the shell. This lets you store a sequence of commands in a file for repeated use. The terms *shell program* and *command file* are often used in place of *shell script*. See *executable file*.

**software**

Software comprises instructions and programs that cause the computer to carry out certain tasks.

**source code**

The uncompiled version of a program written in a programming language such as C or Pascal. The source code must be translated into machine language by a program known as a compiler before the computer can execute the program.

**special character**

Special characters are any characters in a character set that are not letters or numbers. The normal character set of a computer, i.e. the characters that can be output to the screen or printer and represented internally by a corresponding bit pattern, includes punctuation marks, mathematical function characters and a series of commercial special characters. Country-specific special characters are, for example, the German umlaut forms. See also *ASCII* and *control characters*.

**special file**

A file used as an interface to an I/O device (e.g. a user terminal, disk drive, tape cartridge drive or printer). The special file passes the minor device number identifying the device involved to the device driver in the kernel.

**SPOOL**

Acronym for **S**imultaneous **P**eripheral **O**perations **O**n **L**ine. The function of a spooler is to organize the output of print files sent simultaneously by a number of applications or users to the same printer.

**standard error**

If you do not want error messages to be output to the printer or screen, you can redirect standard error to a file by specifying an argument in the form *2> file*. Error messages are then written to the specified file.

### **standard input**

Data transfer in a data processing system using a special input device, e.g. a keyboard or scanner, which accepts incoming data, converts it into the internal format and passes it to main memory. Under Reliant UNIX, standard input is an open file that is normally connected directly to the keyboard.

Standard input for a command is normally passed from the keyboard to this file and then to the shell. You can redirect standard input to come from a file instead of from the keyboard by specifying an argument in the form `< file`. The command will then read its input from the specified file.

### **standard output**

The process of writing data from main memory to a peripheral device. Under Reliant UNIX, it is an open file that is normally connected directly to a primary output device such as the screen. Standard output from the computer is normally placed in this file first and then passed to the output device. Standard output under Reliant UNIX is normally written to the screen. You can redirect standard output into a file instead of to the printer or screen by specifying an argument in the form `> file`. Output is then written to the file you specify.

### **startup**

See also *login*.

### **string**

Any number of characters stored or processed as a single unit. A string may be a logical string (e.g. a text) or a physical string (e.g. a data block to be input or output).

### **string variable**

A sequence of characters that can be the value of a shell variable. See *variable*.

### **subdirectory**

A directory within a directory.

**system administrator**

The person responsible for monitoring a data processing system and its components. System administrators usually have the following organizational tasks:

- data backup
- data protection
- work process scheduling
- coordination of system operations

**tape cartridge**

Also known as *streamer tape*. This is an off-line storage medium similar to magnetic tape, except that the tape is enclosed in a rigid case that also contains parts of the transport mechanism. Tapes are commonly 1/4" (6.4 mm) wide and 300 ft. (91.5 m) long, or 1/2" (12.7 mm) wide and 560 ft. (171 m) long.

**teleprocessing**

Communication via the telephone or telex network, special teleprocessing networks (e.g. DATEX, ISDN), radio links or satellite transmission. In these networks, computers are interconnected with terminals or other computers and can exchange data over long distances.

**terminal**

An I/O device that enables you to transfer data to a CPU over long distances and/or receive data from this CPU. A terminal consists of a keyboard, a monitor and sometimes a printer.

**text editor**

A program for creating, modifying and deleting text. Two of the editors provided by Reliant UNIX are *ced* and *vi*. See *line editor* and *screen editor*.

**text formatter**

A program that prepares a text file for printed output. To make use of a text formatter, your file must contain some special commands for structuring the final copy. These special commands specify the font and font size, margin settings, new paragraphs, lists and tables, position of figures, etc. Two text formatters available under Reliant UNIX are *nroff* and *troff*.

### track

A track on a floppy disk forms a circle around the center of the floppy disk. The individual data bits are distributed along this track. Each track is divided into an equal number of sectors. The number of tracks and sectors on a floppy disk depends on the format. A double-sided floppy disk has the same number of tracks and sectors on each side. Two tracks in the same location on opposite sides of a floppy disk are known as cylinders.

### tty

Originally the abbreviation for **Teletypewriter**, a device for sending and receiving messages across telex lines. Today, it is generally used to denote any user terminal.

### user

Users are people who carry out their own or their company's tasks on a computer. A privileged user is a user who has special tasks and who therefore has precedence over others when competing for the use of system facilities. See *system administrator*.

### user-defined variable

A named variable to which a user has assigned a value. See *variable*.

### utility

Utilities are system programs and are therefore part of the operating system. They usually provide functions that make it easier, faster, more economical or more convenient for the user to work with the computer. For example, there are utilities that help you perform tests or sort and merge data.

### variable

An operand, identified by its name, which is stored in a field and whose value may change. In terms of the shell, a variable is a symbol that stands for a particular string of characters. You can use variables when interacting with the shell and within a shell script. A shell script may contain two types of variable: positional parameters and keyword parameters. For more information on keyword parameters, see chapter 5 *The shell*.

### virtual terminal

See *pseudo terminal*.



**working directory**

See *current directory*.



---

# List of figures

|                                                                        |     |
|------------------------------------------------------------------------|-----|
| Figure 1: What happens to your input . . . . .                         | 22  |
| Figure 2: Sample file system . . . . .                                 | 28  |
| Figure 3: The directories under home . . . . .                         | 33  |
| Figure 4: Absolute path name of the /home/darkstar directory . . . . . | 37  |
| Figure 5: Relative path name from darkstar to project . . . . .        | 39  |
| Figure 6: Relative path name from darkstar to addresses . . . . .      | 40  |
| Figure 7: Output produced by the ls -l command . . . . .               | 57  |
| Figure 8: Displaying a file in a vi window . . . . .                   | 168 |
| Figure 9: The vi screen . . . . .                                      | 172 |
| Figure 10: Screen-oriented positioning . . . . .                       | 205 |
| Figure 11: Scrolling forward . . . . .                                 | 209 |
| Figure 12: Scrolling backward . . . . .                                | 210 |
| Figure 13: The ced screen . . . . .                                    | 261 |



---

# List of tables

|           |                                                                |    |
|-----------|----------------------------------------------------------------|----|
| Table 1:  | Summary of the passwd command . . . . .                        | 20 |
| Table 2:  | Login procedure . . . . .                                      | 20 |
| Table 3:  | Important keys and key combinations . . . . .                  | 23 |
| Table 4:  | Summary of the login procedure . . . . .                       | 26 |
| Table 5:  | Important keys and key combinations . . . . .                  | 26 |
| Table 6:  | Summary of the pwd command . . . . .                           | 35 |
| Table 7:  | Sample path names . . . . .                                    | 41 |
| Table 8:  | Summary of the cd command . . . . .                            | 43 |
| Table 9:  | The meaning of permissions for files and directories . . . . . | 44 |
| Table 10: | Summary of the chmod command . . . . .                         | 50 |
| Table 11: | Summary of the mkdir command . . . . .                         | 53 |
| Table 12: | Summary of the ls command . . . . .                            | 58 |
| Table 13: | Summary of the cat command . . . . .                           | 60 |
| Table 14: | Summary of the echo command . . . . .                          | 60 |
| Table 15: | Summary of the rm command . . . . .                            | 62 |
| Table 16: | Summary of the rmdir command . . . . .                         | 63 |
| Table 17: | Basic commands for using files . . . . .                       | 64 |
| Table 18: | Summary of the cat command . . . . .                           | 67 |
| Table 19: | Some of the pg commands . . . . .                              | 68 |
| Table 20: | Summary of the pg command . . . . .                            | 70 |
| Table 21: | Summary of the pr command . . . . .                            | 72 |
| Table 22: | Summary of the cp command . . . . .                            | 75 |
| Table 23: | Summary of the mv command . . . . .                            | 78 |
| Table 24: | Summary of the wc command . . . . .                            | 80 |

**List of tables**

---

|                                                                      |     |
|----------------------------------------------------------------------|-----|
| Table 25: Summary of the chown command . . . . .                     | 81  |
| Table 26: Summary of the diff command . . . . .                      | 84  |
| Table 27: Summary of the grep command . . . . .                      | 86  |
| Table 28: Summary of the sort command . . . . .                      | 88  |
| Table 29: Command summary . . . . .                                  | 90  |
| Table 30: Summary of the ps command . . . . .                        | 94  |
| Table 31: Summary of the kill command . . . . .                      | 97  |
| Table 32: Summary of the nohup command . . . . .                     | 99  |
| Table 33: Summary of commands for process control . . . . .          | 99  |
| Table 34: Summary of the spell command . . . . .                     | 106 |
| Table 35: Summary of the metacharacters for file name generation . . | 113 |
| Table 36: Summary of the shell special characters . . . . .          | 116 |
| Table 37: Summary of the dl shell script . . . . .                   | 119 |
| Table 38: Summary of the birthday shell script . . . . .             | 121 |
| Table 39: Summary of the whoson shell script . . . . .               | 122 |
| Table 40: Summary of the get.num shell script . . . . .              | 123 |
| Table 41: Summary of the show.param shell script . . . . .           | 124 |
| Table 42: Keyword parameters (default variables) . . . . .           | 125 |
| Table 43: Summary of the mknum shell script . . . . .                | 128 |
| Table 44: Summary of the num.please shell script . . . . .           | 128 |
| Table 45: Summary of the t shell script . . . . .                    | 129 |
| Table 46: Summary of the log.time shell script . . . . .             | 130 |
| Table 47: Summary of the gbdy shell script . . . . .                 | 132 |
| Table 48: Summary of the mv.file shell script . . . . .              | 136 |
| Table 49: Summary of the search shell script . . . . .               | 141 |
| Table 50: Summary of the mv.ex shell script . . . . .                | 143 |

|                                                                       |     |
|-----------------------------------------------------------------------|-----|
| Table 51: Summary of the set.term shell script . . . . .              | 146 |
| Table 52: Entering new text . . . . .                                 | 174 |
| Table 53: Appending text . . . . .                                    | 175 |
| Table 54: Inserting text . . . . .                                    | 176 |
| Table 55: Inserting a blank line . . . . .                            | 178 |
| Table 56: Summary of the commands for creating text . . . . .         | 181 |
| Table 57: Positioning the cursor . . . . .                            | 182 |
| Table 58: Moving the cursor using the arrow keys . . . . .            | 182 |
| Table 59: Summary of the positioning commands . . . . .               | 184 |
| Table 60: Deleting a character . . . . .                              | 185 |
| Table 61: Summary of basic commands . . . . .                         | 189 |
| Table 62: Moving the cursor to the start or end of a line . . . . .   | 190 |
| Table 63: Searching for strings . . . . .                             | 193 |
| Table 64: Line-oriented positioning . . . . .                         | 195 |
| Table 65: Using w to move the cursor word by word . . . . .           | 196 |
| Table 66: Using W and e to move the cursor word by word . . . . .     | 197 |
| Table 67: Using E, b and B to move the cursor word by word . . . . .  | 199 |
| Table 68: Sentence-oriented positioning . . . . .                     | 201 |
| Table 69: Paragraph-oriented positioning . . . . .                    | 203 |
| Table 70: Screen-oriented positioning . . . . .                       | 205 |
| Table 71: Summary of cursor positioning commands . . . . .            | 208 |
| Table 72: Moving the cursor to a specific line . . . . .              | 211 |
| Table 73: Command repetition . . . . .                                | 213 |
| Table 74: Summary of additional cursor positioning commands . . . . . | 215 |
| Table 75: Deleting text using BACKSPACE . . . . .                     | 216 |
| Table 76: Deleting text using <code>CTRL</code> w . . . . .           | 217 |

**List of tables**

---

|                                                                     |     |
|---------------------------------------------------------------------|-----|
| Table 77: Undoing a command . . . . .                               | 218 |
| Table 78: Deleting a sentence . . . . .                             | 220 |
| Table 79: Deleting a paragraph . . . . .                            | 220 |
| Table 80: Summary of commands for deleting text . . . . .           | 222 |
| Table 81: Replacing text using r . . . . .                          | 223 |
| Table 82: Substituting text using s . . . . .                       | 224 |
| Table 83: Changing text . . . . .                                   | 226 |
| Table 84: Summary of commands for overwriting text . . . . .        | 228 |
| Table 85: The temporary buffer p . . . . .                          | 229 |
| Table 86: Summary of temporary buffers and the y command . . . .    | 230 |
| Table 87: Summary of commands for cutting and pasting text . . . .  | 232 |
| Table 88: Summary of special commands . . . . .                     | 232 |
| Table 89: Replacing text . . . . .                                  | 237 |
| Table 90: Summary of the line editor commands . . . . .             | 238 |
| Table 91: Summary of line editor commands for quitting vi . . . . . | 239 |
| Table 92: Summary of special options . . . . .                      | 242 |
| Table 93: ced summary . . . . .                                     | 280 |
| Table 94: ed summary . . . . .                                      | 281 |
| Table 95: sed summary . . . . .                                     | 282 |
| Table 96: awk relational operators . . . . .                        | 288 |
| Table 97: awk regular expressions . . . . .                         | 292 |
| Table 98: awk special variables . . . . .                           | 297 |
| Table 99: awk built-in arithmetic functions . . . . .               | 300 |
| Table 100: awk built-in string functions . . . . .                  | 301 |
| Table 101: The awk function getline . . . . .                       | 317 |
| Table 102: awk printf conversion characters . . . . .               | 320 |



|                                                                                         |     |
|-----------------------------------------------------------------------------------------|-----|
| Table 103: Summary of the mail command (send mode) . . . . .                            | 371 |
| Table 104: Summary of the mail command (read mode) . . . . .                            | 374 |
| Table 105: Summary of the mail command options . . . . .                                | 375 |
| Table 106: Components of the message header . . . . .                                   | 381 |
| Table 107: Changing the message header . . . . .                                        | 382 |
| Table 108: Quitting mailx . . . . .                                                     | 383 |
| Table 109: msglist arguments . . . . .                                                  | 384 |
| Table 110: Summary of the mailx command options . . . . .                               | 391 |
| Table 111: Illegal commands in the .mailrc file . . . . .                               | 393 |
| Table 112: Summary of tar command for archiving files and directories                   | 401 |
| Table 113: Summary of cpio command for archiving files and directories                  | 402 |
| Table 114: Summary of tar command for reading archive contents . . .                    | 404 |
| Table 115: Summary of cpio command for reading archive contents . .                     | 405 |
| Table 116: Summary of tar command for extracting files and directories                  | 407 |
| Table 117: Summary of cpio command for extracting files and directories                 | 408 |
| Table 118: Summary of dd command for file conversion . . . . .                          | 410 |
| Table 119: Summary of doscp for copying files between Reliant UNIX and<br>DOS . . . . . | 413 |
| Table 120: Summary of commands for archiving and conversion . . .                       | 414 |
| Table 121: The number of writable sides on a floppy disk . . . . .                      | 422 |
| Table 122: Storage capacity of floppy disks . . . . .                                   | 423 |
| Table 123: Special files for formatting floppy disks with dosformat . . .               | 426 |
| Table 124: Summary of the floppy disk formatting commands: format .                     | 427 |
| Table 125: Summary of the floppy disk formatting commands: dosformat                    | 427 |
| Table 126: Summary of the tape cartridge processing commands: mt .                      | 431 |
| Table 127: Displaying SCSI devices with gettypes . . . . .                              | 433 |

**List of tables**

---

Table 128: Commands that support regular expressions . . . . . 450

Table 129: Simple regular expressions . . . . . 451

Table 130: Metacharacters in regular expressions . . . . . 455

Table 131: Extended regular expressions . . . . . 456

Table 132: Precedence of operators in regular expressions . . . . . 457

Table 133: Examples of simple regular expressions . . . . . 457

Table 134: Examples of extended regular expressions . . . . . 457

Table 135: ISO 646 character set . . . . . 458

---

# Related publications

To order manuals, please contact your local dealer.

A collection of manuals that are often required is also available online. The online documentation can also be obtained on CD, or can be accessed via the home page of Siemens Nixdorf Informationssysteme AG on the World Wide Web.

- [1] Reliant UNIX  
**Operating Manual (hardware-specific)**

*Target group*

System administrators, users

*Contents*

This manual describes how the hardware is installed, and how the system is placed in service and operated. The contents of the manual are dependent on the computer type.

- [2] Reliant UNIX 5.44  
**Reliant UNIX Installation and Operation, RM200, RM300, RM400**  
Installation Guide

*Target group*

System administrators

*Contents*

This manual describes how to install the Reliant UNIX 5.43 operating system both as a new installation and as an update. Also documented are the basics of the operating system and key aspects of using it as well as instructions for installing the software using the SYSADM operator system.

- [3] Reliant UNIX 5.44  
**Reliant UNIX Installation and Operation, RM600-xxx, RM600-E**  
Installation Guide

*Target group*

System administrators

### *Contents*

This manual describes how to install the Reliant UNIX 5.43 operating system both as a new installation and as an update. Also documented are the basics of the operating system and key aspects of using it as well as instructions for installing the software using the SYSADM operator system.

- [4] Reliant UNIX 5.44  
**Documentation Guide**  
**Annotated Overview, Index**

### *Target group:*

Users and system administrators, network users and administrators, Spool users and administrators and programmers.

### *Contents*

The "Documentation Guide" provides an overview of the documentation available for Reliant UNIX as well as the titles of the manuals and their order numbers. As well as a listing of the contents of the most important manuals, an index is provided which sets out the activities covered by the respective manuals and acts as a reference to the manual where the actual activity is documented.

- [5] Reliant UNIX 5.44  
**Commands, User Reference Manual**  
Reference Manual

### *Target group*

Users, system administrators

### *Contents*

This manual is intended for reference purposes. It describes the user commands of the Reliant UNIX 5.44 operating system.

- [6] Reliant UNIX 5.44  
**System Administrator's Guide**

### *Target group*

Users

### *Contents*

This manual provides a basic overview of Reliant UNIX system administration, with information about important system features and the most tasks for the system administrator.

- [7] Reliant UNIX 5.44  
**System Administrator's Reference Manual**  
Reference Manual
- Target group*  
System administrators
- Contents*  
This manual describes the system administrator commands, file formats, device files and procedures for system maintenance in alphabetical order.
- [8] Reliant UNIX 5.44  
**Network Administration**  
System Administrator's Guide
- Target group*  
Network administrators managing UNIX systems in TCP/IP networks
- Contents*  
This manual describes the network administration activities that have to be performed when using the TCP/IP software on Reliant UNIX 5.44 as well as the basic network function (BNU).
- [9] Reliant UNIX 5.44  
**Network Reference Manual**  
Reference Manual
- Target group*  
Network administrators, system administrators
- Contents*  
This manual gives a description in alphabetical order of the commands, C functions, file formats, system maintenance procedures as well as other Reliant UNIX tools for operating networks.
- [10] Reliant UNIX 5.44  
**System Administration and Hardware Configuration Using the SYSADM User Interface**  
System Administrator's Guide
- Target group*  
System administrators

### *Contents*

This manual describes the SYSADM user interface with the integrated configuration tool Config. The Config is available under the menu option Configuration once the operating system has been installed in SYSADM. The user interface provides a user-friendly means of configuring and managing I/O devices.

- [11] Reliant UNIX 5.44  
**Hardware Configuration with Config under SINIX/windows**

### *Target group*

System administrators

### *Contents*

This manual describes the functionality of Config under the SINIX/windows user interface.

- [12] Reliant UNIX 5.44  
**Virtual Disks**  
User Guide

### *Target group*

System administrators

### *Contents*

This manual describes the various types of virtual disk and how they are configured.

- [13] Reliant UNIX 5.44  
**Veritas File System (VxFS) V1.2**  
System Administrator's Guide

### *Target group*

System administrators

### *Contents*

This manual provides an overview of the structure and the features of the Veritas file system (VxFS) and also describes how it is used. In addition, the commands and utilities used with VxFS are documented and detailed explanations of error and diagnostic messages are given.

- [14] Reliant UNIX 5.44  
**Tuning Guide**  
System Administrator's Guide  
*Target group*  
System administrators  
*Contents*  
This manual describes options for improving the system performance.
- [15] Xprint V5.0  
**User and Administrator Guide**  
*Target group*  
Reliant UNIX users and system administrators; Spool administrators  
*Contents*  
This manual provides information on the Spool concept and documents the objects that make up the Spool. In addition, it describes how Spool is installed as well as how it is used via the command interface.
- [16] Xprint V5.0  
**Reference Manual**  
*Target group*  
Reliant UNIX users and system administrators; Spool administrators  
*Contents*  
This manual describes the Spool commands in alphabetical order, and documents the Spool messages, the configuration files for Spool objects as well as the standard Spool data formats. The manual also presents an overview of how the Spool system works.
- [17] SINIX  
**AT&T Spool**  
System Administrator's Guide  
*Target group*  
System administrators  
*Contents*  
This manual describes how to set up and manage the LP print service.

## Related publications

---

- [18] SINIX/windows User Environment V3.1  
**Documentation Overview**

*Target group*

SINIX/windows users

*Contents*

The manual provides an overview of the SINIX/Windows documentation.

- [19] SINIX/windows User Environment V3.0 (CDE)  
**User's Guide**  
User Guide

*Target group*

Users with basic knowledge of Reliant UNIX

*Contents*

This manual describes the underlying features of the SINIX/windows desktop and explains how to work with the desktop and the desktop applications.

- [20] SINIX/windows User Environment V3.0 (SINIX Desktop)  
**Introduction to Handling and Configuration**  
User Guide

*Target group*

Users with basic knowledge of Reliant UNIX

*Contents*

This manual provides an introduction to operating and configuring the SINIX/windows graphical user interface. It describes the components of the SINIX/windows desktop, how to use the user interface, and inter-active configuration of the desktop.

- [21] SINIX/windows User Environment V3.0 (CDE)  
**Advanced User and System Administrator Guide**  
System Administrator's Guide

*Target group*

System administrators

*Contents*

This manual covers advanced tasks in customizing the appearance and behavior of the Desktop.



- [22] SINIX/windows User Environment V3.1 (CDE)

**CDE Enhancements**

User Guide

*Target group*

Users and system administrators

*Contents*

The book provides an overview of the additional functionality that Siemens Nixdorf Informationssysteme AG and Triteal Corporation have added to the Common Desktop Environment (CDE).

- [23] SINIX/windows User Environment V3.0 (SINIX Desktop)

**Guide for Experts and System Administrators**

System Administrator's Guide

*Target group*

System administrators

*Contents*

This manual discusses the concepts underlying SINIX/windows, such as the client-server model, and configuration of a client using resources. It also describes the primary clients.

- [24] SINIX/windows User Environment

**Clients Reference Manual**

Reference Manual

*Target group*

System administrators

*Contents*

This manual provides the experienced user with a comprehensive overview of the SINIX/windows clients and the resources used by the clients.

- [25] SINIX/windows User Environment

**XDCL Desktop Configuration Language**

Reference Manual

*Target group*

System administrators

*Contents*

This manual provides an overview of the XDCL language, with which the experienced user can configure the SINIX/windows desktop.

- [26] SINIX  
**UNIX System Security**  
User Guide

*Target group*

Users, system administrators, system programmers

*Contents*

This manual describes the user classes of a UNIX system, provides an overview of the security risks in a UNIX system, and explains the ways of reducing these risks.

- [27] SINIX  
**High Availability Guide**  
User Guide

*Target group*

System administrators, Systemsicherheits-Beauftragte

*Contents*

The High Availability Guide describes ways of increasing the availability of UNIX systems.

- [28] SINIX  
**OBSERVE**  
User Guide

*Target group*

OBSERVE system administrators and OBSERVE programmers

*Contents*

This manual describes how OBSERVE is installed and configured in single-, dual- and multi-computer configurations.

- [29] Reliant UNIX  
**AUDIT V2.0**  
System Administrator Guide

*Target group*

System administrators

*Contents*

This manual describes how AUDIT is installed and configured.

- [30] Reliant UNIX 5.44  
**UFS UNIX File System**

*Target group*  
System administrators

*Contents*

This manual describes the organization, administration, maintenance and consistency checking for the ufs file system.

- [31] Reliant UNIX 5.44  
**Error Diagnosis and Error Handling**

*Target group*  
System administrators, programmers and service technicians

*Contents*

This manual describes potential system errors, how these errors are interpreted, as well as possible measures for dealing with errors. In addition, the user interfaces available for system diagnostics are described.

- [32] IOCS V3.0  
**Administration, Configuration and Programming of Printers**

*Target group*  
System administrators, application programmers

*Contents*

This manual describes how to manage, configure, install and program printers using IOCS and also outlines the compatible control sequences for printer programming. It provides information on programming with the IOCS but does not include programming information.

- [33] Reliant UNIX, Windows NT  
**RAIDmaster**  
System Administrator's Guide

*Target group*  
System administrators

*Contents*

The manual describes the configuration of the RAIDmaster-controllers.



---

# Index

- " (sh) 114
- & 97
- ' (sh) 114
- \* (sh) 110
- . 38
- .. 38
- .cshrc 357
- .exrc
- vi default settings (vi) 170
- .mailrc
- default settings (mailx) 392
- defaults (mailx) 377
- mailx defaults (mailx) 376
- .profile 149, 357
- .rhosts 345, 354, 357
- /dev/null 133
- /etc/hosts.equiv 345, 354, 357
- < 104
- > 105
- >> 106
- ? (sh) 112
- [] (sh) 112
- \ (sh) 114

## A

- absolute path name 35, 38
- access
  - direct 418
  - sequential 418
- access permissions
  - for access to remote hosts with rcp 354
  - for access to remote hosts with rlogin 345
  - for access to remote hosts with rsh 357
- access protection see permission
- access rights
  - permission lists for forms 441
  - permission lists for users 441

## action

- awk 284, 294
- complex (awk) 296
- simple (awk) 295

## address 465

## alias 359

## aliases 357

## anonymous ftp 351

## application program 9

## archive 465

- extracting directories from archive (cpio) 407
- extracting directories from archive (tar) 406
- extracting files from archive (cpio) 407
- extracting files from archive (tar) 406
- extracting from floppy disk (cpio) 407
- extracting from floppy disk (tar) 406
- extracting from magnetic tape (cpio) 407
- extracting from magnetic tape (tar) 406
- extracting from tape cartridge (cpio) 407
- extracting from tape cartridge (tar) 406
- reading contents (cpio, tar) 403

## archiving

- directory (cpio, tar) 397
- file (cpio, tar) 397
- on floppy disk 417
- on floppy disk (cpio) 401
- on floppy disk (tar) 398
- on magnetic tape 417
- on magnetic tape (cpio) 401
- on magnetic tape (tar) 398
- on tape cartridge 417

- archiving (continued)
  - on tape cartridge (cpio) 401
  - on tape cartridge (tar) 398
- ARGC
  - special variable (awk) 297, 317
- argument 465
  - awk 317
  - file name generation (sh) 110
- ARGV
  - special variable (awk) 297, 317
- array
  - awk 308
  - one-dimensional 308
- ASCII 465
  - converting ASCII file to EBCDIC (dd) 409
  - converting EBCDIC file to ASCII (dd) 409
- assigning a value to a variable 126
- asterisk (sh) 110
- at 445
- awk
  - accumulation 329
  - action 284, 294
  - addition 298
  - ARGC 297, 317
  - argument 317
  - ARGV 297, 317
  - arithmetic 297
  - array 308
  - BEGIN pattern 284, 286, 292
  - break statement 308
  - built-in arithmetic functions 297
  - calling from the command line 285
  - calling with a program file 286
  - comment line 311
  - comparison 336
  - complex action 296
  - composite assignment operator 300
  - continue statement 308
  - control-flow statements 306
  - data type conversion 304
  - decrement 300
  - awk (continued)
    - delete statement 310
    - division 298
    - does not match 287
    - do-while statement 307
    - END pattern 284, 286, 292
    - equal to 287
    - error messages 327
    - escape sequence 290
    - exit statement 308
    - exponent 298
    - field separator 292, 297, 321
    - field separators 313
    - FILENAME 297, 322
    - FNR 297, 314
    - for statement 307, 309
    - FS 297, 313
    - generating form letters 331
    - generating reports 327
    - getline function 314
    - greater than 287
    - greater than or equal to 287
    - gsub function 302
    - if statement 306
    - increment 300
    - index function 302
    - initialization 336
    - input 312
    - input from files 312
    - input from pipes 312
    - length function 302
    - less than 287
    - less than or equal to 287
    - logical AND 293
    - logical OR 293
    - match function 302
    - matches 287
    - multiline record 314
    - multiplication 298
    - negation 293
    - NF 297, 314
    - not equal to 287
    - NR 314
    - OFMT 297

- awk (continued)
  - OFS 297, 321
  - one-dimensional array 308
  - ORS 297, 321
  - output 318
  - output field separators 321
  - output to file 322
  - output to pipe 322
  - pattern 284, 286
  - pattern ranges 294
  - permitted values 336
  - print function 318
  - printf function 318, 319
  - program execution 284
  - program main section 284
  - program structure 284
  - rand function 330
  - record separator 297, 314, 321
  - regular expression 286, 289, 292, 293
  - relational expression 286, 287
  - remaindering 298
  - return statement 311
  - RLENGTH 297, 302
  - RS 297, 314
  - RSTART 297, 302
  - sample applications 326, 327
  - simple action 295
  - simple assignment 300
  - special variables 295, 296
  - split function 302, 310
  - sprintf function 302
  - sub function 302
  - SUBSEP 297
  - substr function 302
  - subtraction 298
  - system function 324
  - type conversion 336
  - user-defined functions 310
  - user-defined variables 295
  - while statement 307
  - word frequency 329
- B**
  - backend 466
  - background
    - executing commands in background 466
  - background process 97, 344, 346
  - backing up
    - on floppy disk (cpio) 401
  - backquotes 109
  - backslash (sh) 114
  - banner 445
  - batch 445
  - batch processing 466
  - baud rate 466
  - BEGIN pattern
    - awk 286, 292
  - bin
    - directory 119
  - block special file see special file
  - Bourne shell see shell
  - brackets
    - square brackets (sh) 112
  - break statement (awk) 308
  - break statement (sh) 146
  - buffer 466
    - keyboard buffer 472
  - buffer contents
    - saving (ced) 262
    - saving (ed) 268
    - saving (sed) 275
    - saving (vi) 235
  - built-in command 13
- C**
  - cancel 445
  - cartridge see tape cartridge
  - case statement (sh) 143
  - cat 58, 60, 65, 106, 445
  - catalog see directory
  - cd 42, 445
  - CD-ROM 466
    - storage capacity 419
  - CD-ROM drive 419

- ced 260
  - calling 260
  - control panel 261
  - creating a file 260
  - creating text 260
  - delete line 263
  - deleting characters 263
  - deleting text 263
  - insert line 263
  - inserting characters 263
  - inserting text 263
  - moving the cursor 262
  - open file 260
  - operating mode 262
  - quitting 264
  - replacing characters 263
  - replacing text 263
  - saving buffer contents 262
  - screen format 261
- character
  - counting characters in a file (wc) 78
  - delete (vi) 185
  - hidden 471
  - inserting (vi) 174, 175
  - search (vi) 193
- character set
  - 7-bit 458
  - ISO 646 458
- character special file see special file
- character string
  - search (vi) 212
- characteristics
  - SINIX system 10
- characters
  - deleting (ced) 263
  - inserting (ced) 263
  - replacing (ced) 263
  - replacing (ed) 267
  - replacing (sed) 273
  - searching (sed) 273
  - searching for (ed) 267
- child process 91
- chmod 43, 46, 48, 118
- chown 80
- command 467
  - built-in command 13
  - entering a command list 102
  - executing in background 466
  - input 102
  - run in the background 97
  - substitution (sh) 128
- command execution
  - foreground 470
- command interpreter see shell
- command language (sh) 101
- command line 467
- command list 102
- command mode
  - vi 173
- command substitution (sh) 109
- commands
  - entering 24
- comment line (sh) 131
- communication functions
  - Sifmllan 341
- compact disc read-only memory see CD-ROM drive
- concept
  - SINIX concepts 10
- connection setup
  - ftp 351
- continue statement (awk) 308
- continue statement (sh) 147
- control character 467
- control-flow statement
  - break statement (awk) 308
  - break statement (sh) 146
  - case statement (sh) 143
  - continue statement (awk) 308
  - continue statement (sh) 147
  - do-while statement (awk) 307
  - exit statement (awk) 308
  - for statement (awk) 307, 309
  - for statement (sh) 134
  - general format (awk) 306
  - if statement (awk) 306
  - if statement (sh) 139



- control-flow statement (continued)
  - until statement (sh) 137
  - while statement (awk) 307
  - while statement (sh) 136
- conversion
  - copy DOS file from DOS floppy disk to SINIX (doscp) 411
  - copy SINIX file to DOS floppy disk (doscp) 411
  - file 409
- converting
  - ASCII file to EBCDIC (dd) 409
  - EBCDIC file to ASCII (dd) 409
- copy
  - local file 355
  - remote directories 356
- copy file
  - with ftp 351
- copying a file (cp) 73
- correcting input errors 22
- counting
  - characters in a file (wc) 78
  - words in a file (wc) 78
- cp 73, 75, 445
- cpio 401
- current directory 34, 467
- cursor 468
  - moving (ced) 262
  - positioning (vi) 182, 189, 208
- cut 446
- D**
- daemons
  - network 357
- data entry see input
- data type
  - conversion (awk) 304
- date 446
- dd 409
- debugging (sh) 147
- delete statement (awk) 310
- deleting a directory (rmdir) 62
- deleting a file (rm) 61
- delimiter 468
- density
  - DD 423
  - floppy disk 423
  - HD 423
- device driver 468
- device number 468
- diff 81, 446
- directory 30, 469
  - archiving (cpio, tar) 397
  - bin 119
  - change (cd) 42
  - changing permissions (chmod) 48
  - creating (mkdir) 52
  - current directory 34, 467
  - deleting (rmdir) 62
  - extracting from archive (cpio) 407
  - extracting from archive (tar) 406
  - hierarchical structure 30
  - hierarchy 475, 480
  - home 31
  - home directory 34, 471
  - listing contents (ls) 53
  - listing permissions (ls) 45
  - login directory 31, 32, 473
  - organization 51
  - permissions 475
  - root directory 30, 478
  - subdirectory 30
  - system directory 30
- directory name 30
- directory structure
  - organization 51
- disk
  - cylinder 468
  - track 482
- disk drive
  - integrated 420
  - off-line 420
- display
  - active sessions 359
  - status of hosts 360
- display active sessions 359
- displaying file contents 64
- displaying file contents (cat) 65

- displaying file contents page by page (pg) 67
- displaying SCSI devices 432
- DOS 469
  - copy DOS file from DOS floppy disk to SINIX (doscp) 411
  - copy SINIX file to DOS floppy disk (doscp) 411
- doscp 411
- dosformat 426
  - special file 426
- double quote (sh) 114
- do-while statement (awk) 307
- drive
  - CD-ROM 419
  - floppy disk 418
  - for magnetic tape 420
  - for WORM disks 420
  - hard disk 420
  - tape cartridge drive 419
- duplex 469
- E**
- EBCDIC
  - converting ASCII file to EBCDIC (dd) 409
  - converting EBCDIC file to ASCII (dd) 409
- echo 58, 60, 446
- ed 264, 446
  - creating file 264
  - creating text 264
  - deleting a line 268
  - displaying text 266
  - ed script 269
  - editing text 266
  - input mode 264, 265
  - inserting text 265
  - open file 264
  - quitting 269
  - replacing characters 267
  - replacing text 267
  - saving a file (ed) 268
  - saving buffer contents 268
- ed (continued)
  - saving text (ed) 268
  - searching for characters 267
  - searching for text 267
  - substituting text 267
  - undo previous command 268
- editor 481
  - creating text (ced) 260
  - creating text (ed) 264
  - creating text (vi) 172
  - delete character (vi) 185
  - delete line (ced) 263
  - delete line (sed) 272
  - delete line (vi) 221
  - delete paragraph (vi) 220
  - delete sentence (vi) 220
  - delete word (vi) 219
  - deleting a line (ed) 268
  - deleting characters (ced) 263
  - deleting text (ced) 263
  - display 471
  - displaying text (ed) 266
  - ed script (ed) 269
  - editing text (ed) 266
  - editing text (sed) 272
  - editing text in a pipeline (sed) 277
  - full-screen 259
  - functionality (sed) 270
  - insert line (ced) 263
  - insert line (sed) 273
  - inserting characters (ced) 263
  - inserting characters (vi) 174, 175
  - inserting text (ced) 263
  - inserting text (ed) 265
  - inserting text (sed) 273
  - inserting text (vi) 174, 175
  - line 259, 472
  - noninteractive editor (sed) 270
  - quitting (ced) 264
  - quitting (ed) 269
  - quitting (vi) 187, 239
  - redirecting standard output (sed) 276
  - replacing characters (ced) 263

- editor (continued)
  - replacing characters (ed) 267
  - replacing characters (sed) 273
  - replacing text (ced) 263
  - replacing text (ed) 267
  - replacing text (sed) 273
  - saving buffer contents (ced) 262
  - saving buffer contents (ed) 268
  - saving buffer contents (sed) 275, 276
  - saving buffer contents (vi) 235
  - saving text (ced) 262
  - saving text (ed) 268
  - saving text (sed) 275, 276
  - saving text (vi) 235
  - search and replace text (vi) 237
  - searching for characters (ed) 267
  - searching for characters (sed) 273
  - searching for text (ed) 267
  - searching for text (sed) 273
  - sed script (sed) 271
  - substituting text (ed) 267
  - substituting text (sed) 273
  - undo previous command (ed) 268
- EDITOR (mailx) 377, 394
- electronic mail 469
- electronic mail see mail
- END pattern
  - awk 286, 292
- entering commands 24
- environment
  - working environment 9
- environment see login environment
- error
  - error redirection 107
- error message 469
- escape character
  - telnet 347
- escape sequence
  - awk 290
- Exabyte cartridge see Video 8 tape cartridge
- execute command on remote host 356
- exit 133
- exit statement (awk) 308
- export 145
- F**
- FACE
  - change language 340
- face 340
- field
  - initialization (awk) 336
- field separator
  - awk 292, 297, 313, 321
- FIFO see pipe
- file 470
  - .profile 149
  - /dev/null 133
  - appending output to a file 106
  - archiving (cpio, tar) 397
  - changing permissions (chmod) 46
  - changing the owner (chown) 80
  - comparing file contents (diff) 81
  - conversion 409
  - conversion from ASCII to EBCDIC (dd) 409
  - conversion from EBCDIC to ASCII (dd) 409
  - copy DOS file from DOS floppy disk to SINIX (doscp) 411
  - copy SINIX file to DOS floppy disk (doscp) 411
  - copy with rcp 355
  - copying (cp) 73
  - counting characters (wc) 78
  - counting lines (wc) 78
  - counting words (wc) 78
  - creating (cat) 58
  - creating (ced) 260
  - creating (echo) 58
  - creating (ed) 264
  - creating (vi) 172
  - deleting (rm) 61
  - direct access 418
- file (continued)

- display contents (vi, view) 241
- displaying contents 64
- displaying contents (cat) 65
- displaying contents page by page (pg) 67
- editing (awk) 283
- editing more than one file (vi) 241
- executable 470
- extracting from archive (cpio) 407
- extracting from archive (tar) 406
- fetch from remote host 348
- file concept 11
- input from a file (awk) 312
- link 51
- listing permissions (ls) 45
- merge contents (sort) 86
- moving (mv) 75
- name 470
- named pipe 51
- open (ced) 260
- open (ed) 264
- ordinary 28, 474
- output to file (awk) 322
- permissions 475
- permissions (chmod) 43
- preparing contents for printing (pr) 70
- reading in (vi) 236
- reading into a message (mailx) 379
- renaming (mv) 75
- restoring after a system crash (vi) 240
- saving text to a file (ced) 262
- saving text to a file (ed) 268
- saving text to a file (sed) 275
- search for pattern (grep) 84
- search for regular expression (grep) 84
- sending (mail) 370
- sending (mailx) 379
- sequential access 418
- sort contents (sort) 86
- file (continued)
  - special file 51
  - structure 28
  - writing text to a file (vi) 235
- file concept 11
- file names 29
- file system 470
  - organization 51
  - path name 35
  - structure 27
- FILENAME
  - special variable (awk) 297, 322
- filter 15, 107, 470
- finger 362
- flformat 425
- floppy disk 470
  - 3 1/2" 422
  - 5 1/4" 422
  - address 422
  - block 422
  - density 423
  - determining the storage capacity 424
  - double-sided or single-sided 422
  - format 422
  - formatting 425, 426
  - gross storage capacity 423
  - handling 421
  - inserting into drive 425
  - label area 472
  - net storage capacity 423
  - potential storage capacity 423
  - preformatted 423
  - reading archive contents (cpio) 405
  - reading archive contents (tar) 403
  - real storage capacity 423
  - removing from drive 425
  - removing write protection from a 3 1/2" floppy disk 424
  - sector 422, 478
  - size 422
  - storage capacity of a formatted floppy disk 423
- floppy disk (continued)

- storage capacity of an unformatted floppy disk 423
- tpi 423
- track 422
- tracks per inch 423
- write protection 424
- floppy disk drive 418
  - access LED 425
  - floppy disk release button 425
  - inserting floppy disk 425
  - removing floppy disk 425
- FNR
  - special variable (awk) 297, 314
- folder (mailx) 395
- for statement (awk) 307, 309
- for statement (sh) 134
- foreground
  - command execution in the foreground 470
- format 425
- formatting
  - floppy disk 425, 426
- FS
  - special variable (awk) 297, 313
- ftp 351
  - abort transfer 354
  - anonymous 351
  - connection setup 351
  - interactive session 351
  - list of remote files 352
- ftp session
  - quit 353
- ftpd daemon 351
- full-duplex 469
- function
  - getline function (awk) 314
  - gsub function (awk) 302
  - index function (awk) 302
  - length function (awk) 302
  - match function (awk) 302
  - print function (awk) 318
  - printf function (awk) 318, 319
  - rand (awk) 330
- function (continued)
  - split function (awk) 302, 310
  - sprintf function (awk) 302
  - sub function (awk) 302
  - substr function (awk) 302
  - system function (awk) 324
  - user-defined (awk) 310
- G**
  - garbage can (sh) 133
  - get 352
  - getline function (awk) 314
  - gettypes 432
  - GID 92
    - effective 92
    - real 92
  - grep 84, 114, 446
  - group
    - permission 45
  - gsub function (awk) 302
- H**
  - half-duplex 469
  - hard disk 471
    - integrated 420
    - off-line 420
    - sector 478
  - hard disk drive 420
  - here document (sh) 131
  - hierarchical structure of the file system 30
  - HOME
    - variable 151
  - home directory 31, 34, 471
  - host
    - check in network 363
    - inaccessible in network 365
  - host names 359
  - host status
    - display 360

## Index

---

### I

- I/O see output, input
- if statement (awk) 306
- if statement (sh) 139
- index function (awk) 302
- information
  - about network 359
  - display on user 362
- initialization
  - field (awk) 336
  - variable (awk) 336
- input 471
  - awk 312
  - command 102
  - command lists 102
  - from a file (awk) 312
  - from a pipe (awk) 312
  - redirecting standard input 14, 104
- input errors
  - correcting 22
- input mode
  - ed 264
  - vi 173
- input/output 471
- Internet addresses 359

### J

- job number 344, 346

### K

- kernel 9
- keyboard input 21
  - correcting input errors 22
  - important key combinations 23
- keyword parameters (sh) 124
- kill 96, 446

### L

- label area 472
- length function (awk) 302
- lex 446
- line
  - counting lines in a file (wc) 78
  - delete (ced) 263

- line (continued)
  - delete (sed) 272
  - delete (vi) 221
  - deleting (ed) 268
  - insert (ced) 263
  - insert (sed) 273
  - inserting (ed) 265
- line addressing
  - vi 236
- link 51
- list
  - defined hosts 359
- local file
  - copy 355
- local system 472
- log in to remote host
  - via Sifmllan user interface 341
  - with rlogin 342
  - with telnet 345
- log out 472
- logging in 17
- logging out 25
- login 472
  - login procedure 20
  - problems during login 20
- login directory 32, 473
- login environment 469
  - modifying 149
- login name
  - logging in 18
- loop (sh) 134
- lowercase 21
- lp 446
- lpr 446
- lpstat 446
- ls 45, 53, 58, 446

**M**

- magnetic tape 473
  - reading archive contents (cpio) 405
  - reading archive contents (tar) 403
- magnetic tape drive 419, 420
- mail 367, 446, 469
  - aliases (mailx) 396
  - change default mailbox (mailx) 387
  - checking for incoming messages (mail) 374
  - closing a message (mailx) 376
  - default mailbox (mail) 373
  - default mailbox (mailx) 386
  - defaults (mailx) 376
  - deleting messages (mail) 373
  - deleting messages (mailx) 388
  - editing a message (mailx) 377
  - electronic signature (mailx) 382
  - ending a message (mail) 368
  - incorporate message from mailbox into reply (mailx) 379
  - integrating text in a message (mailx) 378
  - keeping a copy of messages (mailx) 380
  - list messages (mailx) 386, 387
  - message header (mailx) 381
  - message title (mailx) 377
  - msglist argument (mailx) 384
  - options 375
  - options (mailx) 391
  - quit (mailx) 383
  - quitting 373
  - quitting (mailx) 390
  - quitting receive mode (mailx) 390
  - quitting send mode (mailx) 383
  - reading a file into a message (mailx) 379
  - reading messages (mail) 371, 374
  - reading messages (mailx) 385
  - receiving (mail) 371
  - receiving messages (mail) 371
- mail (continued)
  - receiving messages (mailx) 383
  - replying to messages (mailx) 389
  - saving message (mailx) 388
  - saving messages (mail) 373
  - send (mailx) 376
  - send message (mail) 368
  - send message to user (mail) 368
  - sending (mail) 368, 369, 370
  - sending (mailx) 379
  - sending a message (mail) 368
  - sending a message to yourself (mail) 368
  - sending files (mail) 370
  - sending messages (mail) 369, 370
  - sending messages (mailx) 376
  - sending messages to several people (mail) 369
  - sending to several people (mail) 369
  - setting environment variables (mailx) 394, 395, 396
  - switching to other mail files (mailx) 387
  - tilde escape (mailx) 376, 377, 378, 379
  - undeliverable mail (mail) 369
- mail see message
- MAILCHECK (mail) 374
- mailx 375, 447
- mailx see mail
  - message
- major device number 468
- match function (awk) 302
- merge
  - file contents (sort) 86
- message
  - checking for incoming messages (mail) 374
  - closing (mailx) 376
  - default mailbox (mail) 373
  - default mailbox (mailx) 386
  - deleting (mail) 373

- message (continued)
    - deleting (mailx) 388
    - editing (mailx) 377
    - electronic signature (mailx) 382
    - ending (mail) 368
    - incorporate from mailbox into
      - reply (mailx) 379
    - integrating text (mailx) 378
    - keeping a copy (mailx) 380
    - list (mailx) 386, 387
    - message header (mailx) 381
    - msglist argument (mailx) 384
    - reading (mail) 371, 374
    - reading (mailx) 385
    - reading a file into a message (mailx) 379
    - receiving (mail) 371
    - receiving (mailx) 383
    - replying to (mailx) 389
    - saving (mail) 373
    - saving (mailx) 388
    - send (mailx) 376
    - sending (mail) 368, 369
    - sending files (mail) 370
    - sending to a user (mail) 368
    - sending to several people (mail) 369
    - sending to yourself (mail) 368
    - switching to other mail files (mailx) 387
    - title (mailx) 377
    - undeliverable (mail) 369
  - metacharacter
    - asterisk (sh) 110
    - question mark (sh) 112
  - metacharacters 13, 353
    - escaping (sh) 113
    - square brackets (sh) 112
  - metacharacters (sh) 110
  - mget 353
  - minor device number 468
  - mkdir 52, 447
  - mode 473
    - append mode 465
    - ced 262
    - command mode 467
    - ed 265
    - input mode 471
    - insert mode 472
    - vi 173
  - modem 473
  - moving
    - file (mv) 75
  - mput 353
  - msglist argument (mailx) 384
  - mt 429
  - mv 75, 78, 447
- N**
- Native Language System 17
  - network 473
  - network functions
    - errors 365
  - Network Information Service (NIS) 361
  - network management master 361
    - information about 362
  - network status 359
  - NF
    - special variable (awk) 297, 314
  - NIS 361
  - NLS 17
  - nohup 98, 447
  - NR
    - special variable (awk) 297, 314
  - numbering system
    - binary 474
    - hexadecimal 474
    - octal 474



**O**

## objects

- permission lists 441

## OFMT

- special variable (awk) 297

## OFS

- field separator (awk) 321

- special variable (awk) 297, 321

## operating system 474

## operator

- addition (awk) 298

- composite assignment operator (awk) 300

- decrement (awk) 300

- division (awk) 298

- does not match (awk) 287

- equal to (awk) 287

- exponent (awk) 298

- greater than (awk) 287

- greater than or equal to (awk) 287

- increment (awk) 300

- less than (awk) 287

- less than or equal to (awk) 287

- logical AND (awk) 293

- logical OR (awk) 293

- matches (awk) 287

- multiplication (awk) 298

- negation (awk) 293

- not equal to (awk) 287

- precedence in regular expression 457

- remaindering (awk) 298

- simple assignment (awk) 300

- subtraction (awk) 298

## option 474

## options

- mail 375

## ordinary file 28

## ORS

- record separator (awk) 321

- special variable (awk) 297, 321

## outfolder (mailx) 395

## output 471, 474

- appending to a file 106

- awk 318

- pipe (awk) 322

- redirecting standard error 14, 107

- redirecting standard output 14, 104, 105

- redirecting standard output (cat) 58

- redirecting standard output (echo) 58

- redirecting standard output (sed) 276

- to file (awk) 322

**P**

## paragraph

- delete (vi) 220

## parameter 474

- positional parameter 476

- positional parameters (sh) 120, 129

- special parameters (sh) 122

## parameters

- special parameters (sh) 124

## parent process 91

## parity 475

## passwd 19, 20

## password 475

- assigning 19

- changing 19

- defining 19

- logging in 18

## PATH

- variable 151

## path name 35, 475

- absolute path name 35, 38, 465

- relative path name 38, 40, 477

## pattern 449, 475

- awk 284

- BEGIN pattern (awk) 286, 292

- END pattern (awk) 286, 292

- pattern ranges (awk) 294

## Index

---

- pattern (continued)
  - regular expression (awk) 286, 289, 293
  - relational expression (awk) 286, 287
  - search (vi) 212
  - search in a file (grep) 84
  - types (awk) 286
- permission
  - default 44
  - execute permission 44
  - file 46
  - group 45
  - l bit (locking bit) 44
  - other system users 45
  - r bit 44
  - read permission 44
  - s bit 44, 93
  - t bit (sticky bit) 44
  - user 45
  - w bit 44
  - write permission 44
  - x bit 44
- permission lists 441
- permissions 475
  - changing (chmod) 43, 46, 48
  - directory 48
  - file (chmod) 43
  - listing (ls) 45
  - specifying in octal format (chmod) 49
- personal computer 476
- pg 67, 447
- PID 92
- ping 363
- pipe 15, 107, 476
  - input from a pipe (awk) 312
  - named pipe 51
  - output to pipe (awk) 322
- pipeline 15, 107, 476
  - editing text in a pipeline (sed) 277
- positional parameter (sh) 120, 129
- pr 70, 447
- preparing file contents for printing (pr) 70
- print function (awk) 318
- printer 476
- printf function (awk) 318, 319
- process 476
  - aborting 96
  - background process 97
  - child process 91
  - control 91, 98
  - GID 92
  - group ID 92
  - identification 92
  - obtaining information on
    - processes 93
  - parent process 91
  - PID 92
  - priority 95
  - process concept 11
  - send signal to a process 96
  - suspending 95
  - UID 92
  - user ID 92
- process concept 11
- process ID 92
- process priority 95
- profile see .profile
- program 477
  - application program 9
  - system program 9
  - user program 9
- program structure
  - awk 284
- programming construct
  - shell script 130
- programming language
  - awk 283
- prompt 477
  - PS1 152
- protection bit see permission
- ps 93, 447
- PS1
  - variable 152
- pseudo terminal 477

put 352  
pwd 34, 447

## Q

question mark (sh) 112  
quitting mail 373

## R

rand  
    built-in arithmetic function (awk) 330  
rcp 355  
    access permissions 354  
read 126  
record  
    awk 314  
record (mailx) 395  
record separator  
    awk 297, 314, 321  
regular expression 16, 449, 477  
    awk 286, 289, 292  
    extended 456  
    metacharacters 456  
    overview 450  
    precedence 457  
    search in a file (grep) 84  
    simple 451  
relational expression  
    awk 286, 287  
relative path name 38, 40  
remote directories  
    copy 356  
remote host  
    log in 342  
remote system 478  
removable disk storage see disk drive  
renaming  
    file (mv) 75  
return code (sh) 133  
return statement (awk) 311  
RLENGTH  
    special variable (awk) 297, 302  
rlogin 342  
    access permissions 345

rlogin connection  
    interrupt 344  
    terminate 344  
rm 61, 447  
rmdir 62, 63, 64, 447  
root directory 30, 478  
RS  
    special variable (awk) 297, 314  
rsh 356  
    access permissions 357  
RSTART  
    special variable (awk) 297, 302  
rusers 359  
rusersd 360  
rwho 359  
rwhod 360

## S

saving  
    message (mailx) 388  
    messages (mail) 373  
    to magnetic tape 417  
    to tape cartridge 417  
script  
    creating a shell script 117  
    ed script (ed) 269  
    executing a shell script 118  
    programming constructs in shell scripts 130  
    sed script (sed) 271  
    shell script 117  
script see shell  
SCSI devices  
    displaying 432  
searching for a pattern in a file (grep) 84  
searching for a regular expression in a file (grep) 84  
searching for a string in a file (grep) 84  
sed 270  
    delete line 272  
    editing text 272  
    editing text in a pipeline 277  
    functionality 270

- sed (continued)
  - insert line (sed) 273
  - inserting text 273
  - redirecting standard output 276
  - replacing characters 273
  - replacing text 273
  - saving a file 275
  - saving buffer contents 275
  - saving text 275, 276
  - searching for characters 273
  - searching for text 273
  - sed script 271
  - substituting text 273
- sentence
  - delete (vi) 220
- session on remote host
  - terminate 343
- Shell
  - important key combinations 23
- shell 12, 478
  - /dev/null 133
  - appending output to a file 106
  - assigning a value to a variable 126
  - asterisk 110
  - backquotes 109
  - backslash 114
  - break statement 146
  - case statement 143
  - command input 102
  - command language 101
  - command list 102
  - command substitution 109, 128
  - comment line 131
  - continue statement 147
  - correcting input errors 22
  - creating a script 117
  - debugging 147
  - double quote 114
  - error redirection 107
  - escaping metacharacters 113
  - executing a script 118
  - export variable 145
  - file name generation 110
  - filter 107
  - for statement 134
  - function of the shell 21
  - garbage can 133
  - here document 131
  - if statement 139
  - keyword parameters 124
  - loop 134
  - lowercase 21
  - metacharacters 110, 112
  - modifying login environment 149
  - pipe 107
  - pipeline 107
  - positional parameters 120, 129
  - programming 117
  - programming constructs in shell scripts 130
  - redirecting input 104
  - redirecting output 104, 105
  - redirecting standard error 107
  - redirecting standard input 104
  - redirecting standard output 104, 105
  - return code 133
  - script 117
  - shell script 479
  - single quote 114
  - special parameters 122, 124
  - square brackets 112
  - testing conditions 141
  - until statement 137
  - uppercase 21
  - variable 151
  - variables 120
  - while statement 136
- shell built-in see built-in command
- shell metacharacters 357
- Sifmllan interface
  - call 340
- Sifmllan user interface 340
- signal
  - number 96
  - send to a process 96

- single quote (sh) 114
  - sleep 95
  - sort 86, 448
    - file contents (sort) 86
  - special character 479
  - special command see built-in command
  - special file 51, 432, 479
    - dosformat 426
  - special parameters (sh) 122
  - spell 105, 106, 448
  - split function (awk) 302, 310
  - SPOOL 479
  - sprintf function (awk) 302
  - standard error 479
    - redirecting 14, 107
  - standard input 480
    - redirecting 14, 104
  - standard output 480
    - redirecting 14, 104, 105
    - redirecting (cat) 58
    - redirecting (echo) 58
  - storage capacity
    - 1/4" tape cartridge 419
    - CD-ROM 419
    - formatted floppy disk 423
    - gross storage capacity of a floppy disk 423
    - net storage capacity of a floppy disk 423
    - potential storage capacity of a floppy disk 423
    - real storage capacity of a floppy disk 423
    - unformatted floppy disk 423
    - Video 8 tape cartridge 419
  - storage device
    - CD-ROM drive 419
    - floppy disk drive 418
    - hard disk drive 420
    - magnetic tape drive 419
    - tape cartridge drive 419
    - WORM drive 420
  - storage medium see floppy disk, magnetic tape, tape cartridge storage medium see floppy disk, tape cartridge
  - store see save
  - storing
    - on floppy disk 417
  - streamer tape see tape cartridge
  - string 480
    - search in a file (grep) 84
    - substitute (ed) 267
    - substitute (sed) 273
  - structure
    - awk program (awk) 284
    - file 28
    - SINIX system 9
  - stty 150, 448
  - sub function (awk) 302
  - subdirectory 30
  - SUBSEP
    - special variable (awk) 297
  - substr function (awk) 302
  - system
    - local 472
    - remote 478
  - system directory 30
  - system function (awk) 324
  - system program 9
  - system users
    - permission 45
- T**
- tape
    - erasing 430
  - tape cartridge 419, 481
    - container 428
    - erasing 430
    - handling 428
    - processing (mt) 429
    - reading archive contents (cpio) 405
    - reading archive contents (tar) 403
    - storage capacity 419
    - write protection 428
  - tape cartridge drive 419

- tar 398
- tar format 421, 425
- tee 148
- teleprocessing 481
- telnet 345
  - escape character 347
- TERM
  - variable 152
- terminal 481, 482
  - configuring 150
  - logging in 18
  - setting options 150
  - virtual 477
- test 141
- text
  - copy to temporary buffer (vi) 230
  - copying (vi) 229, 230
  - creating (ced) 260
  - creating (ed) 264
  - creating (vi) 172
  - deleting (ced) 263
  - deleting (ed) 268
  - deleting (sed) 272
  - deleting (vi) 185, 216, 236
  - displaying (ed) 266
  - editing (awk) 283
  - editing (ed) 266
  - editing (sed) 272
  - editing in a pipeline (sed) 277
  - formatter 481
  - inserting (ced) 263
  - inserting (ed) 265
  - inserting (sed) 273
  - inserting (vi) 174, 175
  - line numbers (vi) 211
  - moving (vi) 229
  - overwriting (vi) 223
  - replacing (ced) 263
  - replacing (ed) 267
  - replacing (sed) 273
  - saving (ced) 262
  - saving (ed) 268
  - saving (sed) 275, 276
  - saving (vi) 235

- text (continued)
  - search (ed) 267
  - search (vi) 193, 212
  - search and replace (vi) 237
  - searching (sed) 273
  - substitute (ed) 267
  - substitute (sed) 273
  - substituting (vi) 223
- tty see terminal
- type
  - conversion (awk) 336

## U

- UID 92
  - effective 92
  - real 92
- uname 448
- until statement (sh) 137
- uppercase 21
- user 482
  - login name 473
  - permission 45
- user program 9
- utility 482

## V

- variable 482
  - ARGC (awk) 297, 317
  - ARGV (awk) 297, 317
  - assigning a value 126
  - EDITOR (mailx) 377, 394
  - export 145
  - FILENAME (awk) 297, 322
  - FNR (awk) 297, 314
  - folder (mailx) 395
  - FS (awk) 297, 313
  - HOME 151
  - initialization (awk) 336
  - MAILCHECK (mail) 374
  - NF (awk) 297, 314
  - NR (awk) 314
  - OFMT (awk) 297
  - OFS (awk) 297, 321
  - ORS (awk) 297, 321

- variable (continued)
  - outfolder (mailx) 395
  - PATH 151
  - positional parameter 476
  - PS1 152
  - record (mailx) 395
  - RLENGTH (awk) 297, 302
  - RS (awk) 297, 314
  - RSTART (awk) 297, 302
  - shell variable 151
  - special (awk) 295, 296, 297
  - string 480
  - SUBSEP (awk) 297
- variable (continued)
  - TERM 152
  - user-defined 482
  - user-defined (awk) 295
  - user-defined (sh) 126
  - VISUAL (mailx) 377, 394
- variables
  - keyword parameters (sh) 124
- variables (sh) 120
- vi 167, 448
  - .exrc 170
  - automatic indentation 171
  - calling 169
  - changing lowercase to uppercase 234
  - changing uppercase to lowercase 234
  - command mode 173
  - command summary 254
  - copy text to temporary buffer 230
  - copying text 230
  - correcting transposed characters 229
  - creating a file 172
  - creating text 172
  - default settings 169
  - delete character 185
  - delete line 221
  - delete paragraph 220
  - delete sentence 220
  - delete word 219
- vi (continued)
  - deleting text 185, 216, 236
  - displaying the operating mode 171
  - displaying the options 169
  - editing more than one file 241
  - exiting input mode 175
  - input mode 173
  - inserting characters 174, 175
  - inserting text 174, 175
  - joining lines together 233
  - line addressing 236
  - line editor commands 234
  - line numbers 211
  - line wrap margin 170
  - login environment 169
  - moving around a file 208
  - moving text 229
  - operating mode 173
  - overwriting characters 223
  - overwriting text 223
  - positioning the cursor 182, 189, 208
  - quitting 187, 239
  - reading in a file 236
  - repeat last command 232
  - restoring a file after a system crash 240
  - returning temporarily to the shell 234
  - saving a file (vi) 235
  - saving buffer contents 235
  - saving text 235
  - screen format 172
  - search and replace text 237
  - search for character 193
  - searching for a character string 212
  - searching for a pattern 212
  - searching for text 212
  - substituting characters 223
  - substituting text 223
  - undoing the last command 218
  - what vi does 167

## Index

---

Video 8 tape cartridge  
    erasing 430  
    storage capacity 419  
view 241  
VISUAL (mailx) 377, 394

## W

wc 78, 448  
while statement (awk) 307  
while statement (sh) 136  
who 448  
word  
    counting words in a file (wc) 78  
    delete (vi) 219  
working directory see current directory  
working environment 9  
WORM drive 420  
write once read many see WORM  
drive  
write protection  
    floppy disk 424  
    on 1/4" tape cartridges 428  
    on Video 8 tape cartridges 428  
    removing from a 3 1/2" floppy disk  
        424

## Y

ypcat 362  
ypmatch 362  
ypwhich 362



Siemens Nixdorf  
Informationssysteme AG  
User Documentation OEC ES EP 5  
81730 Munich

**Fax: (++49) 89636-48717**

Comments  
Suggestions  
Corrections

---

Submitted by

---

Comments on Reliant UNIX 5.44  
User's Guide

---

U25903-J-Z145-1-7600





Siemens Nixdorf  
Informationssysteme AG  
User Documentation OEC ES EP 5  
81730 Munich

**Fax: (++49) 89636-48717**

Comments  
Suggestions  
Corrections

---

Submitted by

---

Comments on Reliant UNIX 5.44  
User's Guide

---

U25903-J-Z145-1-7600





Siemens Nixdorf  
Informationssysteme AG  
Training Center, Consulting Service  
81730 Munich  
Germany

**Fax: (089) 636-52003**

# Courses Consulting Self-tuition media

The Siemens Nixdorf Training Center offers consulting for in-service training, courses and self-tuition media on almost all aspects of information technology, either on your premises or at a Training Center near you. Write to us for information and course programs.

---

Submitted by

---

send me information/course programs on the following topics

---

U25903-J-Z145-1-7600





Siemens Nixdorf  
Informationssysteme AG  
Training Center, Consulting Service  
81730 Munich  
Germany

**Fax: (089) 636-52003**

# Courses Consulting Self-tuition media

The Siemens Nixdorf Training Center offers consulting for in-service training, courses and self-tuition media on almost all aspects of information technology, either on your premises or at a Training Center near you. Write to us for information and course programs.

---

Submitted by

---

send me information/course programs on the following topics

---

U25903-J-Z145-1-7600

