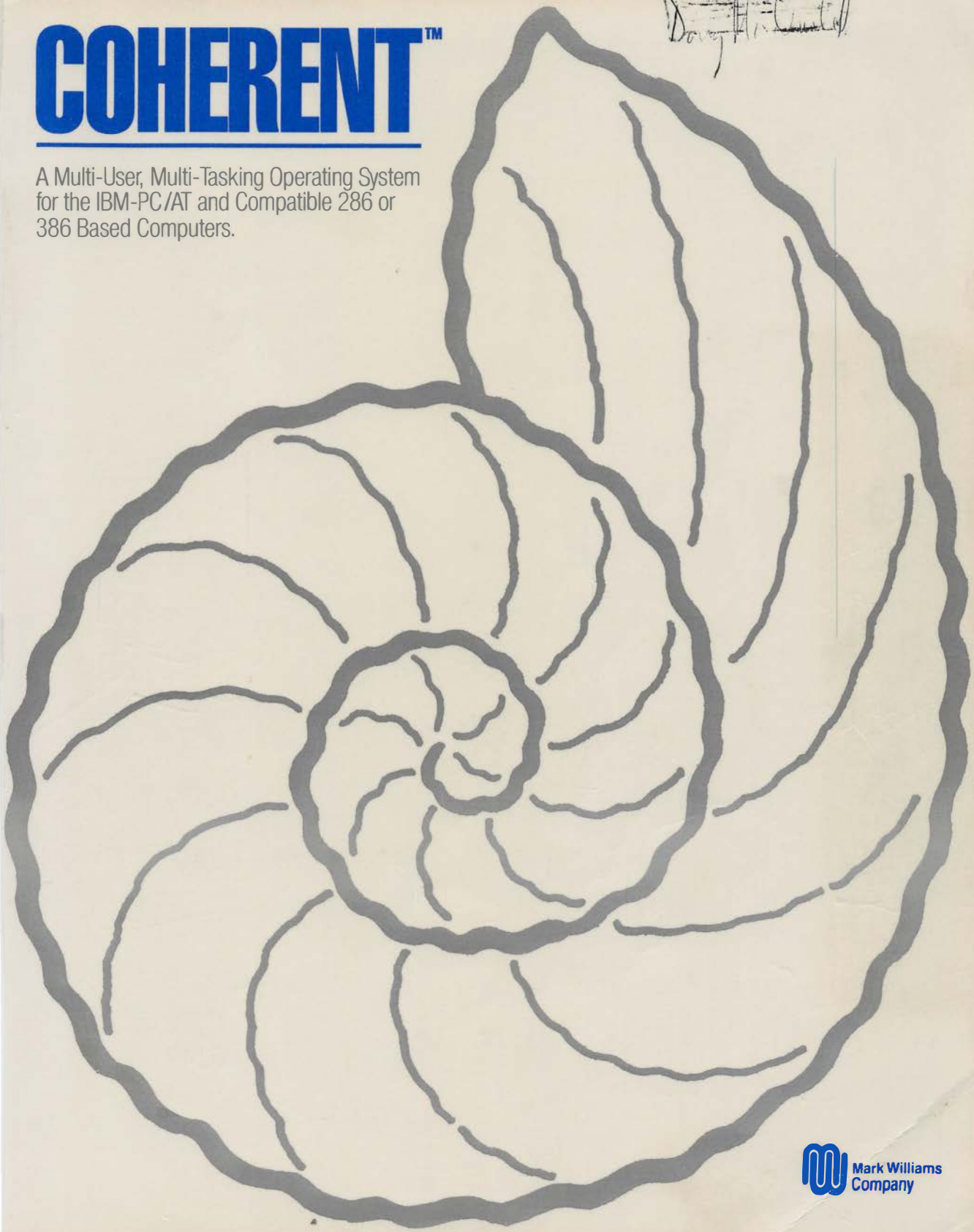
COHERENT™

A Multi-User, Multi-Tasking Operating System for the IBM-PC/AT and Compatible 286 or 386 Based Computers.



Mark Williams Company

Need Tapes?

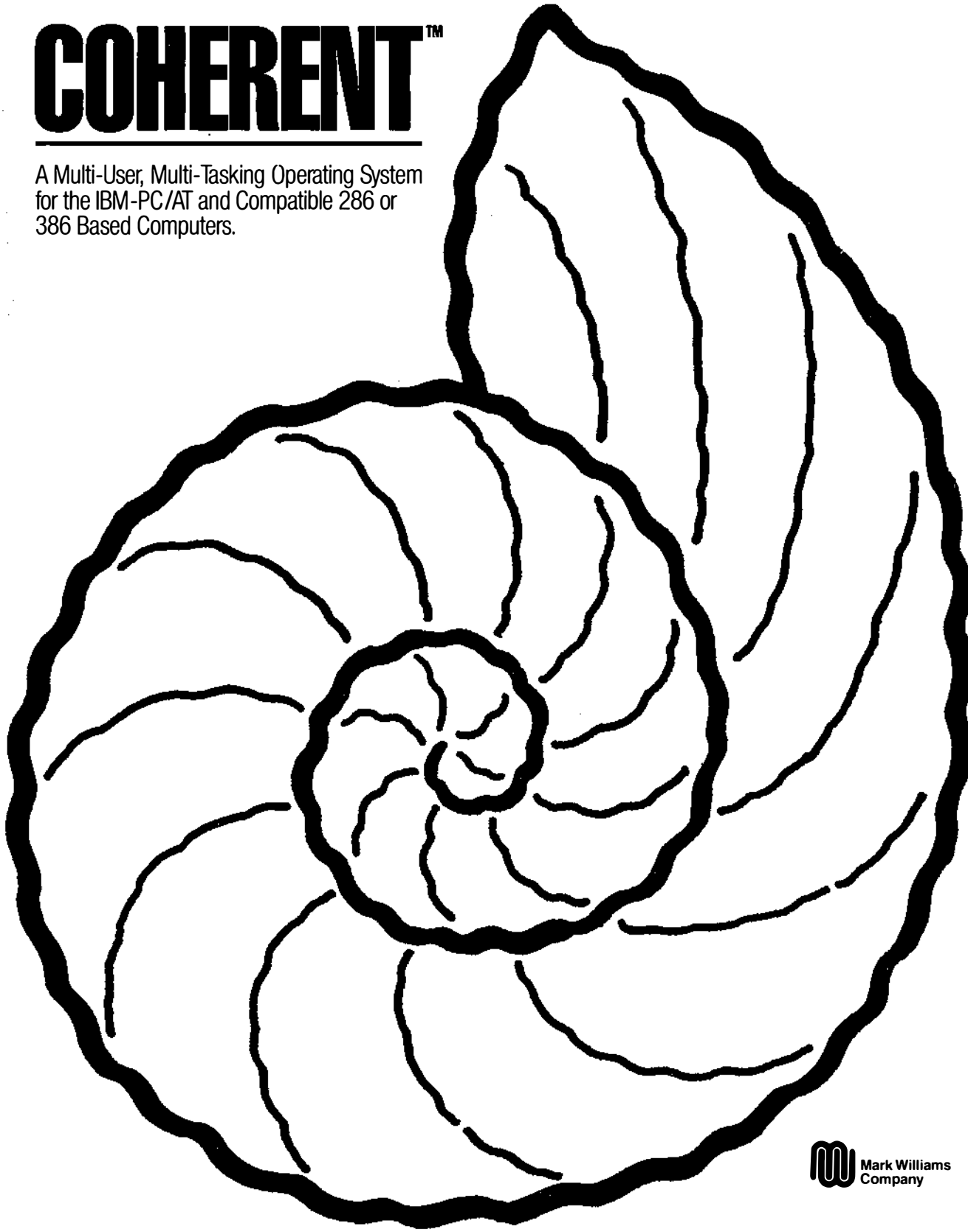
Contact Your Dealer Or Call:

1-800-432-5858

012-309

COHERENT™

A Multi-User, Multi-Tasking Operating System for the IBM-PC/AT and Compatible 286 or 386 Based Computers.



Mark Williams Company

Copyright © 1982, 1991 by Mark Williams Company.

All rights reserved.

This publication conveys information that is the property of Mark Williams Company. It shall not be copied, reproduced or duplicated in whole or in part without the express written permission of Mark Williams Company. Mark Williams Company makes no warranty of any kind with respect to this material and disclaims any implied warranties of merchantability or fitness for any particular purpose.

COHERENT is a trademark of Mark Williams Company. UNIX is a trademark of AT&T. MS-DOS is a copyright of Microsoft Corporation. All other products are trademarks or registered trademarks of the respective holders.

Revision 6 Printing 5 4 3 2 1

Published by Mark Williams Company, 60 Revere Drive, Northbrook, Illinois 60062.

E-mail:	uunet!mwc!support	(Technical Support)
	support@mwc.com	
	uunet!mwc!sales	(General Information)
	sales@mwc.com	
BIX:	join mwc	
CompuServ:	76256,427	

This manual was written under the COHERENT operating system, using the MicroEMACS and **ed** text editors. Text formatting was performed by the COHERENT edition of the **troff** text formatter, using its PostScript output function. Typesetting of this manual, from the table of contents through the index, was performed by one script written in the COHERENT Korn shell. Camera-ready copy was printed on a Hewlett-Packard LaserJet IIP printer using the Pacific Page PostScript cartridge.

Printed in the U.S.A.

Preface

COHERENT is the work of a large number of exceptionally talented people. The development of a multi-user, multi-tasking operating system is a daunting task. Creating COHERENT took an enormous effort by all involved. The system and manual are dedicated to those who dedicated themselves to COHERENT.

These people include the following:

Jay Alter
Bob Beals
Luddyne Blue
Fred Butzen
Allan Cornish
Ella Dashevsky
Tom Duff
Charles Fiterman
Johann George
Walter Grogan
Randall Howard
Nancy Kenston
Irene Lee
Karen McBride
Tim Murphy
Steve Ness
Frank Pfeiffer
Hal Snyder
Robert Swartz
Diane Tracey
La Monte Yarroll
Riyaz Asaria
James Behr
Barry Bowen
Henry Cejtin
Roger Critchlow
Stephen Davis
Mark Epstein
Charles Forsyth
Daniel Glasser
Robert Hemedinger
Owen Jacobsen
J.T. Kittridge
Dave Levine
Scott Moody
Asia Negron
Ciaran O'Donnell
Norma Reyes
Michael Spertus
Angus Telfer
Rico Tudor
Jim Yonan
Norman Bartek
Chris Berrios
Denise Buirge
David Conroy
Richard Critchlow
Mimi Diaz
Michael Farley
Kim Fruin
Michael Griffin
Scott Hermes
Mary Karabatsos
William Lederer
Jeanne Lewis
Esther Munoz
Gerson Negron
Douglas Peterson
Vladimir Smelyansky
Julie Stewart
Trevor Thompson
Bill Witt

Table of Contents

1. Introduction . . . 1
- Hardware Requirements . . . 1
- How To Use This Manual . . . 1
  - The Lexicon . . . 2
- User Registration and Reaction Report . . . 2
- Technical Support . . . 2

2. Using the COHERENT System . . . 3
- How Do I Begin? . . . 3
  - Logging in . . . 3
  - Special Terminal Keys . . . 4
  - Try Some COHERENT Commands . . . 5
  - Giving Commands to COHERENT . . . 6
  - help, man: Help with Commands . . . 7
  - Shutting Down COHERENT and Rebooting . . . 7
  - Logging Out . . . 9
- Working With Files and Directories . . . 9
  - File Names . . . 10
  - Introduction to Directories . . . 10
  - Path Names . . . 11
  - ls, lc: Listing Your Directory . . . 12
  - cat: Print Contents of a File . . . 14
  - more: List Files on the Screen . . . 14
  - mkdir: Create a Directory . . . 14
  - cd: Change Directory . . . 15
  - pwd: Print Working Directory . . . 15
  - mv, cp: Move and Copy Files . . . 16
  - rm, rmdir: Remove Files and Directories . . . 18
  - du, df: How Much Space? . . . 19
  - ln: Link Files . . . 19
  - File Permissions . . . 20
  - chmod: Change File Permissions . . . 21
  - Creating and Mounting a File System . . . 22
  - fdformat: Format a Diskette . . . 22
  - mkfs: Create a File System . . . 23
  - mount: Mount a File System . . . 23
  - Using a Newly Mounted File System . . . 24
  - umount: Unmount a File System . . . 24
  - fsck: Check a File System . . . 25
  - Devices, Files, and Drivers . . . 25
  - Character-Special Files . . . 26
  - tty Processing . . . 26
  - A Tour Through the File System . . . 26
  - General File System Layout . . . 27
  - /bin . . . 27
  - /dev . . . 27
  - /drv . . . 27
  - /etc . . . 28

/lib . . . 28
/usr . . . 28
/u . . . 28
Files: Conclusion . . . 29
Introduction to COHERENT Commands . . . 29
The Shell . . . 29
Redirecting Input and Output . . . 29
Pipes . . . 30
Superuser . . . 30
Manipulating Text Under COHERENT . . . 31
MicroEMACS: Text Screen Editor . . . 31
pr, prps, lpr: Print Files . . . 33
nroff, troff: Text Formatters . . . 33
Miscellaneous Commands . . . 35
who: Who Is On the System . . . 35
write: Electronic Dialogue . . . 35
mail: Send an Electronic Letter . . . 36
msgs: Cumulative Message Board . . . 38
grep: Find Patterns in Text Files . . . 38
date: Print the Date . . . 39
passwd: Change Your Password . . . 39
stty: Change Terminal Behavior . . . 40
Scheduling Commands For Regular Execution . . . 41
Managing Processes . . . 42
ps: List Active Processes . . . 42
kill: Signal Processes . . . 43
Programming Under COHERENT . . . 44
Basic Steps in COHERENT Programming . . . 44
Create the Program Source . . . 44
cc: Compile the Program . . . 45
m4: Macro Processing . . . 46
make: Build Larger Programs . . . 46
db: Debug the Program . . . 46
Administering the COHERENT System . . . 47
Adding a New User . . . 47
System Security . . . 48
Passwords . . . 48
File Protection . . . 48
Encryption . . . 49
Dumping and Saving Files . . . 49
Back-ups Using ustar . . . 49
Back-ups Using dump . . . 50
Restoring Information . . . 51
Dump and Restoring Using cpio . . . 52
System Accounting . . . 52
ac: Login Accounting . . . 53
sa: Processing Accounting . . . 54
Conclusion . . . 56
3. Introduction to the awk Language . . . 57
Using awk . . . 57
Program Structure . . . 57
Records and Fields . . . 58
Command Line Arguments . . . 60

Printing with awk . . . 61
Printing Individual Fields . . . 61
Changing the Output Field and Record Separators . . . 61
Printing Predefined Variables . . . 62
Redirecting Output . . . 62
Formatting Output . . . 63
Piping Output . . . 63
awk Pattern Scanning . . . 64
Special Patterns: BEGIN and END. . . . 64
Patterns . . . 64
Arithmetic Relational Expressions . . . 66
Boolean Combinations of Expressions. . . . 66
Pattern Ranges . . . 67
Specifying awk Actions. . . . 67
Functions . . . 67
Assignment of Variables . . . 68
Field Variables. . . . 69
String Concatenation. . . . 70
Arrays. . . . 70
Control Statements . . . 71
if (condition) else . . . 71
while (condition). . . . 71
for . . . 71
break . . . 72
continue . . . 72
next . . . 72
exit . . . 72
For More Information. . . . 73
4. bc Desk Calculator Language . . . 75
Entry and Exit. . . . 75
Example of Simple Use. . . . 75
Simple Statements . . . 76
Numbers with Fractions . . . 79
The Scale of Numbers . . . 79
Addition and Subtraction . . . 80
Scale During Multiplication . . . 80
Setting the Scale of Results . . . 80
Scale for Divisions . . . 81
Scale From Exponentiation . . . 81
What Is the Current Scale? . . . 81
The if Statement. . . . 82
Using the if Statement . . . 82
Comparisons. . . . 82
Grouped Statements . . . 83
Many Statements Per Line. . . . 83
The while Statement . . . 84
Abbreviations in the while Statement . . . 85
The for Statement. . . . 85
Three Parts of the for Statement . . . 86
Similarities Between the for and while Statements . . . 87
Functions in bc . . . 87
Example of Function Use . . . 87
Functions Using Other Functions . . . 88

Functions That Call Themselves . . . 89
The auto Statement . . . 89
Programs in a File . . . 90
Using a Program From a File . . . 90
Using Libraries . . . 90
The bc Library . . . 92
Summary . . . 92
5. The C Language . . . 93
Compiling C Programs under COHERENT . . . 93
Try the Compiler . . . 93
Phases of Compilation . . . 94
Renaming Executable Files . . . 95
Floating-Point Numbers . . . 95
Compiling Multiple Source Files . . . 96
Linking Without Compiling . . . 96
Compiling Without Linking . . . 96
Assembly-Language Files . . . 97
Changing the Size of the Stack . . . 97
Where To Go From Here . . . 97
C for Beginners . . . 97
Programming Languages and C . . . 98
Assembly and High-Level Languages . . . 98
So, What Is C? . . . 98
Structured Programming . . . 99
Writing a C Program . . . 100
A Sample C Programming Session . . . 100
Designing a Program . . . 101
The main() Function . . . 102
Open a File and Show Text . . . 102
Accepting File Names . . . 104
Error Checking . . . 106
Print a Portion of a File . . . 108
Checking for the End of File . . . 110
Polling the Keyboard . . . 111
For More Information . . . 113
Bibliography . . . 113
6. Introduction to ed, Interactive Line Editor . . . 117
Why You Need an Editor . . . 117
Learning To Use the Editor . . . 117
General Topics . . . 117
ed, Files, and Text . . . 118
Creating a File . . . 119
Changing an Existing File . . . 119
Working on Lines . . . 120
Error Messages . . . 120
Basic Editing Techniques . . . 120
Creating a New File . . . 120
Changing a File . . . 122
Printing Lines . . . 123
Abbreviating Line Numbers . . . 124
How Many Lines? . . . 124
Removing Lines . . . 125
Abandoning Changes . . . 126

Substituting Text Within a Line . . . 126
Undoing Substitutions . . . 128
Global Substitutions . . . 129
Special Characters . . . 129
Ranges of Substitution . . . 130
Intermediate Editing . . . 130
Relative Line Numbering . . . 130
Changing Lines . . . 132
Moving Blocks of Text . . . 132
Copying Blocks of Text . . . 134
String Searches . . . 134
Remembered Search Arguments . . . 136
Uses of Special Characters . . . 136
Global Commands . . . 137
Joining Lines . . . 137
Splitting Lines . . . 138
Marking Lines . . . 139
Searching in Reverse Direction . . . 141
Expert Editing . . . 141
File Processing Commands . . . 141
Patterns . . . 143
Matching Many With One Character . . . 144
Beginning and Ending of Lines . . . 144
Replacing Matched Part . . . 145
Replacing Parts of Matched String . . . 145
Listing Funny Lines . . . 147
Keeping Track of Current Line . . . 148
When Current Line Is Changed . . . 148
More About Global Commands . . . 150
Issuing COHERENT Commands Within ed . . . 151
For More Information . . . 151
7. Introduction to lex, the Lexical Analyzer . . . 153
How To Use lex . . . 153
Translating Strings . . . 153
Remove Blanks From Input . . . 154
Trimming Blanks . . . 154
lex Specification Form . . . 154
Simple Form . . . 155
Rules in lex . . . 155
Statements in lex . . . 156
Groups of Statements . . . 157
Using the Same Action . . . 158
Patterns . . . 158
Simple Patterns . . . 159
Classes of Characters . . . 159
Repetition . . . 161
Choices and Grouping . . . 163
Matching Non-Graphic Characters . . . 163
More Patterns . . . 164
Line Context . . . 164
Context Matching . . . 164
Macro Abbreviations . . . 165
Context: Start Rules . . . 166

Separate Contexts . . . 167
More About Writing Actions . . . 169
ECHO . . . 169
Processing Overlapping Strings . . . 169
yylex . . . 170
Header Section . . . 171
Additional Routines . . . 172
Using lex With yacc . . . 172
Summary . . . 173
8. Introduction to the m4 Macro Processor . . . 175
Definitions and Syntax . . . 175
Defining Macros . . . 176
Input Control . . . 178
Output Control . . . 179
String Manipulation . . . 179
Numeric Manipulation . . . 180
COHERENT System Interface . . . 182
Errors . . . 183
For More Information . . . 184
9. The make Programming Discipline . . . 185
How Does make Work? . . . 185
Try make . . . 186
Essential make . . . 187
The makefile . . . 187
Building a Simple makefile . . . 188
Comments and Macros . . . 188
Setting the Time . . . 189
Building a Large Program . . . 189
Command Line Options . . . 190
Other Command Line Features . . . 191
Advanced make . . . 192
Default Rules . . . 192
Source File Path . . . 193
Double-Colon Target Lines . . . 193
Alternative Uses . . . 194
Special Targets . . . 195
Errors . . . 196
Exit Status . . . 196
Where To Go From Here . . . 196
10. Introduction to MicroEMACS . . . 197
What is MicroEMACS? . . . 197
Keystrokes: <ctrl>, <esc> . . . 197
Becoming Acquainted with MicroEMACS . . . 197
Beginning a Document . . . 199
Moving the Cursor . . . 200
Moving the Cursor Forward . . . 200
Moving the Cursor Backwards . . . 200
From Line to Line . . . 200
Repetitive Motion . . . 201
Moving Up and Down by a Screenful of Text . . . 201
Moving to Beginning or End of Text . . . 201
Saving Text and Quitting . . . 202
Killing and Deleting . . . 202

Deleting Vs. Killing . . . 202
Erasing Text to the Right . . . 203
Erasing Text to the Left . . . 203
Erasing Lines of Text . . . 204
Yanking Back (Restoring) Text . . . 204
Quitting . . . 204
Block Killing and Moving Text . . . 204
Moving One Line of Text . . . 204
Multiple Copying of Killed Text . . . 205
Kill and Move a Block of Text . . . 205
Capitalization and Other Tools . . . 206
Capitalization and Lowercasing . . . 206
Transpose Characters . . . 207
Screen Redraw . . . 207
Return Indent . . . 207
Word Wrap . . . 208
Search and Reverse Search . . . 209
Search Forward . . . 209
Reverse Search . . . 210
Cancel a Command . . . 211
Search and Replace . . . 211
Saving Text and Exiting . . . 212
Write Text to a New File . . . 212
Save Text and Exit . . . 213
Advanced Editing . . . 213
Arguments . . . 214
Arguments: Default Values . . . 214
Selecting Values . . . 215
Deleting With Arguments: An Exception . . . 215
Buffers and Files . . . 215
Definitions . . . 215
File and Buffer Commands . . . 216
Write and Rename Commands . . . 216
Replace Text in a Buffer . . . 216
Visiting Another Buffer . . . 217
Move Text From One Buffer to Another . . . 217
Checking Buffer Status . . . 218
Renaming a Buffer . . . 218
Delete a Buffer . . . 219
Windows . . . 219
Creating Windows and Moving Between Them . . . 220
Enlarging and Shrinking Windows . . . 221
Displaying Text Within a Window . . . 221
One Buffer . . . 222
Multiple Buffers . . . 222
Moving and Copying Text Among Buffers . . . 223
Checking Buffer Status . . . 223
Saving Text From Windows . . . 223
Keyboard Macros . . . 223
Creating a Keyboard Macro . . . 224
Execute a Macro Repeatedly . . . 224
Replacing a Macro . . . 225
Renaming a Macro . . . 225

Renaming Macros: A Few Caveats . . . 225
Setting the Initialization Macro . . . 226
Flexible Key Bindings . . . 226
Changing a Keybinding . . . 226
Rebinding Metakeys . . . 227
Save and Restore Keybindings . . . 228
Sending Commands to COHERENT . . . 228
Compiling and Debugging Through MicroEMACS . . . 229
The MicroEMACS Help Facility . . . 230
Where To Go From Here . . . 230
11. nroff, The Text-Formatting Language . . . 231
What is nroff? . . . 231
nroff Input and Output . . . 232
Printing nroff Output . . . 232
nroff Limitations . . . 233
The ms Macro Package . . . 233
Using this Tutorial . . . 234
The -ms Macro Package . . . 234
Text and Commands . . . 235
Command Names . . . 236
Paragraphs . . . 236
Section Headings . . . 241
Title Page . . . 243
Headers and Footers . . . 244
Fonts . . . 245
Special Characters . . . 246
Footnotes . . . 247
Displays and Keeps . . . 247
Other Commands . . . 248
Introducing nroff's Primitives . . . 249
Page Format . . . 249
Breaks . . . 250
Fill and Adjust Modes . . . 250
Defining Paragraphs . . . 252
Centering . . . 253
Tabs . . . 254
Page Breaks . . . 254
Macros and Traps . . . 255
What Is a Macro? . . . 255
Introducing Traps . . . 256
Headers and Footers . . . 257
Macro Arguments . . . 259
Double vs. Single Backslashes . . . 260
Designing and Installing Macros . . . 261
Strings . . . 264
Strings Within Strings . . . 265
Number Registers . . . 266
Incrementing and Decrementing . . . 268
Units of Measurement . . . 271
Conditional Input . . . 273
Environments and Diversions . . . 277
Buffers . . . 280
Headers and Footers . . . 281

More About Fonts . . . 282
Diversions . . . 283
A Footnote Macro . . . 286
Command Line Options . . . 287
For Further Information . . . 289
12. Introduction to the sed Stream Editor . . . 291
Getting to Know sed . . . 291
Getting Started . . . 291
Simple Commands . . . 292
Substituting . . . 293
Selecting Lines . . . 294
p: Print Lines . . . 296
Line Location . . . 298
Add Lines of Text . . . 299
Delete Lines . . . 300
Change Lines . . . 301
Include Lines From a File . . . 302
Quit Processing . . . 303
Next Line . . . 303
Advanced sed Commands . . . 305
Work Area . . . 305
Add to Work Area . . . 305
Print First Line . . . 307
Save Work Area . . . 308
Transform Characters . . . 311
Command Control . . . 312
{ }: Command Grouping . . . 312
!: All But . . . 312
= : Print Line Number . . . 313
Skipping Commands . . . 313
t: Test Command . . . 314
For More Information . . . 315
13. Introducing sh, the Bourne Shell . . . 317
Simple Commands . . . 317
Special Characters . . . 317
Running Commands in the Background . . . 318
Scripts . . . 319
.profile: Login Shell Script . . . 320
Substitutions . . . 320
File Name Substitution . . . 320
Parameter Substitution . . . 322
Shell Variable Substitution . . . 324
Command Substitution . . . 327
Special Shell Variables . . . 327
dot . : Read Commands . . . 328
Values Returned by Commands . . . 328
test: Condition Testing . . . 329
Executing Commands Conditionally . . . 329
Control Flow . . . 330
for: Execute a Loop . . . 330
if: Execute Conditionally . . . 331
while: Execute a Loop . . . 333
until: Another Looping Construct . . . 333

case: Serial Conditional Execution . . . 333
Summary . . . 334
14. UUCP, Remote Communications Utility . . . 335
Contents of This Manual . . . 335
An Overview of UUCP . . . 336
The Programs . . . 336
Directories and Files . . . 337
Attaching a Modem to Your Computer . . . 338
Installing UUCP . . . 340
Setting Up Your Local Site . . . 340
Describing a Remote Site . . . 342
Day and Time of Connection . . . 342
The Chat Script . . . 344
Granting Permissions . . . 345
Setting a Polling Time . . . 347
Sending files via UUCP . . . 348
UUCP Administration . . . 349
Where to Go From Here . . . 350
15. Introduction to yacc . . . 351
Examples . . . 351
Phrases and Parentheses . . . 351
Simple Expression Processing . . . 353
Background . . . 355
LR Parsing . . . 355
Input Specification . . . 355
Parser Operation . . . 356
Form of yacc Programs . . . 356
Rules . . . 357
Definitions . . . 357
User Code . . . 357
Rules . . . 357
General Form of Rules . . . 358
Suggested Style . . . 358
Actions . . . 359
Basic Action Statements . . . 359
Action Values . . . 360
Structured Values . . . 362
Handling Ambiguities . . . 363
How yacc Reacts . . . 364
Additional Control . . . 365
Precedence . . . 365
Error Handling . . . 367
Summary . . . 368
Helpful Hints . . . 368
Example . . . 369
Where to Go From Here . . . 375
16. The Lexicon . . . 377
example . . . Give an example of Mark Williams Lexicon format . . . 378
. . . String-ize operator . . . 379
. . . Token-pasting operator . . . 380
#define . . . Define an identifier as a macro . . . 381
#elif . . . Include code conditionally . . . 382
#else . . . Include code conditionally . . . 383

#endif . . . . . End conditional inclusion of code 383
#if . . . . . Include code conditionally. 383
#ifdef . . . . . Include code conditionally. 383
#ifndef . . . . . Include code conditionally. 384
#include . . . . . Read another file and include it 384
#line. . . . . Reset line number 385
#undef . . . . . Undefine a macro 385
__DATE__. . . . . Date of translation 385
__FILE__ . . . . . Source file name 386
__LINE__ . . . . . Current line within a source file 386
__STDC__. . . . . Mark a conforming translator. 386
__TIME__. . . . . Time source file is translated 387
_exit() . . . . . Terminate a program 387
abort(). . . . . End program immediately 388
abs(). . . . . Return the absolute value of an integer 388
ac . . . . . Summarize login accounting information. 389
access() . . . . . Check if a file can be accessed in a given mode 390
access.h . . . . . Check accessibility 391
acct() . . . . . Enable/disable process accounting 391
acct.h . . . . . Format for process-accounting file 392
accton. . . . . Enable/disable process accounting 393
acos() . . . . . Calculate inverse cosine 393
action.h. . . . . Describe parsing action and goto tables. 394
address 394
aha154x . . . . . Adaptec AHA-154x device driver 395
alarm() . . . . . Set a timer 397
alias . . . . . Set an alias 397
aliases . . . . . File of users' aliases 398
alignment 400
alloc.h. . . . . Define the allocator. 400
ar . . . . . The librarian/archiver 400
ar.h . . . . . Format for archive files. 401
arena 402
argc . . . . . Argument passed to main() 402
argv . . . . . Argument passed to main() 403
array 403
as . . . . . i80X86 assembler. 404
ASCII 423
ascii.h. . . . . Define non-printable ASCII characters 425
asctime() . . . . . Convert time structure to ASCII string 425
asin() . . . . . Calculate inverse sine 426
ASKCC . . . . . Force prompting for CC names 427
assert() . . . . . Check assertion at run time. 427
assert.h. . . . . Define **assert()**. 427
at . . . . . Drivers for hard-disk partitions. 428
at . . . . . Execute commands at given time. 429
atan() . . . . . Calculate inverse tangent 430
atan2() . . . . . Calculate inverse tangent 431
ATclock. . . . . Read or set the AT realtime clock. 431
atof() . . . . . Convert ASCII strings to floating point 431
atoi(). . . . . Convert ASCII strings to integers. 432
atol(). . . . . Convert ASCII strings to long integers. 432
atrun . . . . . Execute commands at a preset time 433

auto . . . . . Note an automatic variable 433
awk . . . . . Pattern-scanning language 434
bad . . . . . Maintain list of bad blocks 436
badscan . . . . . Build bad block list 436
banner . . . . . Print large letters 437
basename . . . . . Strip path information from a file name 437
bc . . . . . Interactive calculator with arbitrary precision 437
bind . . . . . Bind key sequence to editing command 440
bit 440
bit-fields 440
bit map 441
block 441
boot . . . . . Boot block for hard-disk partition/nine-sector diskette 442
boot.fha . . . . . Boot block for floppy disk 444
booting . . . . . How booting works 444
boottime . . . . . File that holds time system was last booted 446
brc . . . . . Perform maintenance chores, single-user mode 446
break . . . . . Exit from shell construct 446
break . . . . . Exit from loop or switch statement 446
brk() . . . . . Change size of data area 447
bsearch() . . . . . Search an array 447
buf.h . . . . . Buffer header 449
buffer 449
build . . . . . Install COHERENT onto a hard disk 450
builtin . . . . . Execute a command as a built-in command 450
byte 450
byte ordering . . . . . Machine-dependent ordering of bytes 450
c . . . . . Print multi-column output 452
C keywords 452
C language 453
C preprocessor 455
cabs() . . . . . Complex absolute value function 456
cal . . . . . Print a calendar 457
calendar . . . . . Reminder service 457
calling conventions 458
calloc() . . . . . Allocate dynamic memory 462
candaddr() . . . . . Convert a daddr_t to canonical format 463
candev() . . . . . Convert a dev_t to canonical format 463
canino() . . . . . Convert an ino_t to canonical format 463
canint() . . . . . Convert an int to canonical format 464
canlong() . . . . . Convert a long to canonical format 464
canon.h . . . . . Portable layout of binary data 464
canshort() . . . . . Convert a short to canonical format 466
cansize() . . . . . Convert an fsize_t to canonical format 466
cantime() . . . . . Convert a time_t to canonical format 466
canvaddr() . . . . . Convert a vaddr_t to canonical format 467
case . . . . . Execute commands conditionally according to pattern 467
case . . . . . Introduce entry in switch statement 467
cast 468
cat . . . . . Concatenate/print files 468
caveat utilitor 469
cc . . . . . Compiler controller 469
cc0 473

cc1 473
cc2 473
cc3 474
cd . . . . . Change directory 474
ceil() . . . . . Set numeric ceiling 474
cgrep . . . . . Pattern search for C source programs 475
char . . . . . Data type 476
chars.h . . . . . Character definitions 477
chase . . . . . Highly amusing video game 477
chdir() . . . . . Change working directory 478
check . . . . . Check file system 478
checklist . . . . . File systems to check when booting COHERENT 478
chgrp . . . . . Change the group owner of a file 479
chmod() . . . . . Change file-protection modes 479
chmod . . . . . Change the modes of a file 480
chown() . . . . . Change ownership of a file 481
chown . . . . . Change the owner of files 482
chroot() . . . . . Change process's root directory 482
ckermit . . . . . Interactive inter-system communication and file transfer . . . 482
clear . . . . . Clear the screen 488
clearerr() . . . . . Present stream status 488
close() . . . . . Close a file 488
clri . . . . . Clear i-node 488
cmp . . . . . Compare bytes of two files 489
COHERENT system calls 489
col . . . . . Remove reverse and half-line motions 491
com . . . . . Device drivers for asynchronous serial lines 491
com1 . . . . . Device driver for asynchronous serial line COM1 493
com2 . . . . . Device driver for asynchronous serial line COM2 494
com3 . . . . . Device driver for asynchronous serial line COM3 494
com4 . . . . . Device driver for asynchronous serial line COM4 495
comm . . . . . Print common lines 495
commands 495
compress . . . . . Compress a file 501
con.h . . . . . Configure device drivers 502
console . . . . . Console device driver 502
const . . . . . Qualify an identifier as not modifiable 506
const.h . . . . . Declare machine-dependent constants 506
continue . . . . . Force next iteration of a loop 506
continue . . . . . Terminate current iteration of shell construct 507
conv . . . . . Numeric base converter 507
core . . . . . Core dump file format 507
cos() . . . . . Calculate cosine 508
cosh() . . . . . Calculate hyperbolic cosine 508
cp . . . . . Copy a file 509
cpdir . . . . . Copy directory hierarchy 510
cpio . . . . . Archiving/backup utility 511
cpp . . . . . C preprocessor 513
creat() . . . . . Create/truncate a file 514
cron . . . . . Execute commands periodically 514
crypt() . . . . . Encryption using rotor algorithm 516
crypt . . . . . Encrypt/decrypt text 516
ct . . . . . Controlling terminal driver 517

ctags . . . . . Generate tags and refs files for vi editor 517
ctime() . . . . . Convert system time to an ASCII string 518
ctype 519
ctype.h . . . . . Header file for data tests 520
curses. . . . . Library of screen-handling functions 521
curses.h . . . . . Define functions and macros in curses library 529
cut. . . . . Select portions of each line of its input 529
CWD . . . . . Current working directory 530
daemon 531
data formats 531
data types 531
date . . . . . Print/set the date and time 532
db . . . . . Assembler-level symbolic debugger 533
dc . . . . . Desk calculator 537
dcheck . . . . . Check directory consistency. 538
dd . . . . . File conversion 539
default . . . . . Default label in switch statement 540
definitions 540
deftty.h . . . . . Define default tty settings 541
deroff . . . . . Remove text formatting control information 541
detab . . . . . Replace tab characters with spaces 542
device drivers 542
df . . . . . Measure free space on disk 543
diff. . . . . Summarize differences between two files 544
diff3 . . . . . Summarize differences among three files 545
dir.h. . . . . Directory format. 545
directory 546
dirent.h. . . . . Define dirent. 546
dirs . . . . . Print the contents of the directory stack 546
disable . . . . . Disable a port 547
div() . . . . . Perform integer division 547
do . . . . . Introduce a loop. 548
dos . . . . . Transfer files to/from an MS-DOS file system 548
double . . . . . Data type. 550
drvld . . . . . Load a loadable driver into memory 551
du . . . . . Summarize disk usage 551
dump . . . . . File-system backup utility 552
dumpdate . . . . . Print dump dates 553
dumpdir . . . . . Print the directory of a dump 553
dumptape.h . . . . . Define data structures used on dump tapes 554
dup() . . . . . Duplicate a file descriptor 554
dup2(). . . . . Duplicate a file descriptor 554
ebcdic.h . . . . . Define manifest constants for EBCDIC characters. 556
echo. . . . . Repeat/expand an argument 556
ed . . . . . Interactive line editor. 556
EDITOR. . . . . Name editor to use by default. 560
egrep . . . . . Extended pattern search. 561
else . . . . . Introduce a conditional statement 562
elvis . . . . . Clone of Berkeley-standard screen editor 562
enable. . . . . Enable a port 570
end 571
endgrent() . . . . . Close group file 571
endpwent() . . . . . Close password file 571

enum . . . . . Declare a type and identifiers 572
ENV . . . . . File read to set environment. 572
environ . . . . . Process environment 573
environmental variables 573
envp. . . . . Argument passed to main() 574
EOF . . . . . Indicate end of a file 574
epson . . . . . Print files on Epson printer 575
errno . . . . . External integer for return of error status 575
errno.h . . . . . Error numbers used by errno() 576
etext. 578
eval . . . . . Evaluate arguments 579
ex . . . . . Berkeley-style line editor. 579
exec . . . . . Execute command directly 580
execl() . . . . . Execute a load module 580
execle() . . . . . Execute a load module 580
execlp() . . . . . Execute a load module 581
executable file 581
execution. 581
execv(). . . . . Execute a load module 582
execve() . . . . . Execute a load module 582
execvp(). . . . . Execute a load module 584
exit . . . . . Exit from a noninteractive shell 584
exit(). . . . . Terminate a program gracefully. 584
exp(). . . . . Compute exponent 585
export. . . . . Add a shell variable to the environment. 586
expr . . . . . Compute a command line expression 586
extern. . . . . Declare storage class 588
fabs() . . . . . Compute absolute value 589
factor . . . . . Factor a number 589
false. . . . . Unconditional failure. 589
fblk.h . . . . . Define the disk-free block 589
fc . . . . . Edit and re-execute one or more previous commands 589
FCEDIT. . . . . Editor used by fc command 590
fclose() . . . . . Close a stream. 590
fcntl() . . . . . Control open files 590
fcntl.h. . . . . Manifest constants for file-handling functions 591
fd . . . . . Floppy disk driver. 591
fd.h . . . . . Declare file-descriptor structure 592
fdformat . . . . . Format a floppy disk 592
fdioctl.h. . . . . Control floppy-disk I/O 593
fdisk. . . . . Hard-disk partitioning utility 593
fdisk.h . . . . . Fixed-disk constants and structures. 595
fdopen(). . . . . Open a stream for standard I/O 595
feof(). . . . . Discover stream status. 596
ferror() . . . . . Discover stream status. 596
fflush() . . . . . Flush output stream's buffer 598
fgetc() . . . . . Read character from stream. 599
fgets() . . . . . Read line from stream 600
fgetw(). . . . . Read integer from stream 601
field 601
file. . . . . Guess a file's type. 602
file. 602
FILE. . . . . Descriptor for a file stream 602

file descriptor 603
file formats 603
fileno() . . . . . Get file descriptor 604
filsys.h . . . . . Structures and constants for super block. 605
filter 605
find . . . . . Search for files satisfying a pattern 605
fixstack . . . . . Change stack allocation 606
float . . . . . Data type 607
floor() . . . . . Set a numeric floor 610
floppy disks 610
fnkey . . . . . Set/print function keys for the console 613
fopen() . . . . . Open a stream for standard I/O 614
for . . . . . Control a loop 615
for . . . . . Execute commands for tokens in list 616
fork() . . . . . Create a new process. 616
fortune . . . . . Print randomly selected, hopefully humorous, text 617
fperr.h . . . . . Constants used with floating-point exception codes. 617
fprintf() . . . . . Print formatted output into file stream 617
fputc() . . . . . Write character into file stream. 618
fputs() . . . . . Write string into file stream 618
fputw() . . . . . Write an integer into a stream 619
fread() . . . . . Read data from file stream. 619
free() . . . . . Return dynamic memory to free memory pool 619
freopen() . . . . . Open file stream for standard I/O 620
frexp() . . . . . Separate fraction and exponent. 621
from . . . . . Generate list of numbers, for use in loop 622
fscanf() . . . . . Format input from a file stream 622
fsck . . . . . Check and repair file systems interactively 623
fseek() . . . . . Seek on file stream 625
fstat() . . . . . Find file attributes 626
ftell() . . . . . Return current position of file pointer 628
ftime() . . . . . Get the current time from the operating system 628
function 628
fwrite() . . . . . Write into file stream 629
fwtable . . . . . Build font-width table 629
gcd() . . . . . Set variable to greatest common divisor 631
general functions 631
getc() . . . . . Read character from file stream 632
getchar() . . . . . Read character from standard input 633
getegid() . . . . . Get effective group identifier 634
getenv() . . . . . Read environmental variable 634
geteuid() . . . . . Get effective user identifier 635
getgid() . . . . . Get real group identifier 635
getgrent() . . . . . Get group file information 635
getgrgid() . . . . . Get group file information, by group name 636
getgrnam() . . . . . Get group file information, by group id 636
getlogin() . . . . . Get login name 636
getopt() . . . . . Get option letter from argv 637
getopts . . . . . Parse command-line options 639
getpass() . . . . . Get password with prompting. 640
getpid() . . . . . Get process identifier. 640
getpw() . . . . . Search password file 640
getpwent() . . . . . Get password file information. 640

getpwnam() . . . . . Get password file information, by name 641
getpwuid() . . . . . Get password file information, by id 641
gets() . . . . . Read string from standard input 642
getty . . . . . Terminal initialization 643
getuid() . . . . . Get real user identifier 644
getw() . . . . . Read word from file stream 644
getwd() . . . . . Get current working directory name 645
GMT 645
gmtime() . . . . . Convert system time to calendar structure 645
goto . . . . . Unconditionally jump within a function 646
grep . . . . . Pattern search 647
group . . . . . Group file format 648
grp.h . . . . . Declare group structure 649
gtty() . . . . . Device-dependent control 649
hard disk 650
hash . . . . . Add a command to the shell's hash table 653
hdioctl.h . . . . . Control hard-disk I/O 653
head . . . . . Print the beginning of a file 653
header files 653
help . . . . . Print concise description of command 655
HOME . . . . . User's home directory 656
hp . . . . . Prepare files for Hewlett-Packard LaserJet printer 656
hpd . . . . . Hewlett-Packard LaserJet printer spooler daemon 657
hpr . . . . . Send file to Hewlett-Packard LaserJet printer spooler 657
hpskip . . . . . Abort/restart current listing on Hewlett-Packard LaserJet . . . 658
hs . . . . . Device driver for polled serial ports 658
hypot() . . . . . Compute hypotenuse of right triangle 660
i-node . . . . . COHERENT system file identifier 661
icheck . . . . . i-node consistency check 661
if . . . . . Introduce a conditional statement 662
if . . . . . Execute a command conditionally 662
IFS . . . . . Characters recognized as white space 662
index() . . . . . Find a character in a string 663
init . . . . . System initialization 663
initialization 664
ino.h . . . . . Constants and structures for disk i-nodes 667
inode.h . . . . . Constants and structures for memory-resident i-nodes 667
install . . . . . Install a software update onto COHERENT 667
int . . . . . Data type 669
interrupt 669
io.h . . . . . Constants and structures used by I/O 669
ioctl() . . . . . Device-dependent control 669
ipc.h . . . . . Definitions for process communications 670
isalnum() . . . . . Check if a character is a number or letter 670
isalpha() . . . . . Check if a character is a letter 670
isascii() . . . . . Check if a character is an ASCII character 671
isatty() . . . . . Check if a device is a terminal 671
iscntrl() . . . . . Check if a character is a control character 671
isdigit() . . . . . Check if a character is a numeral 671
islower() . . . . . Check if a character is a lower-case letter 672
ispos() . . . . . Return if variable is positive or negative 672
isprint() . . . . . Check if a character is printable 672
ispunct() . . . . . Check if a character is a punctuation mark 672

isspace() . . . . . Check if a character prints white space 673
isupper() . . . . . Check if a character is an upper-case letter 673
itom() . . . . . Create a multiple-precision integer. 673
j0(). . . . . Compute Bessel function 674
j1(). . . . . Compute Bessel function 675
jn(). . . . . Compute Bessel function 675
jobs . . . . . Print information about jobs 675
join . . . . . Join two data bases 676
kermit. . . . . Inter-system communication and file transfer 677
keyboard tables . . . . . How to write a keyboard table 680
kill() . . . . . Kill a system process 684
kill. . . . . Signal a process. 685
ksh . . . . . The Korn shell. 685
KSH_VERSION . . . . . List current version of Korn shell. 703
L-devices . . . . . Describe devices used by UUCP 704
l.out.h. . . . . Object file format 704
L.sys . . . . . Format for UUCP site descriptions 705
l3tol() . . . . . Convert file system block number to long integer 708
LASTERROR. . . . . Program that last generated an error 709
lc . . . . . List/categorize files in a directory 709
ld . . . . . Link relocatable object files 710
ldexp(). . . . . Combine fraction and exponent 712
ldiv(). . . . . Perform long integer division 712
let . . . . . Evaluate an expression 713
lex . . . . . Lexical analyzer generator. 713
Lexicon 715
libraries. 717
limits.h . . . . . Define numerical limits 717
lines. . . . . Highly amusing board game. 719
link() . . . . . Create a link 719
linker-defined symbols 720
ln . . . . . Create a link to a file 720
localtime() . . . . . Convert system time to calendar structure 721
log() . . . . . Compute natural logarithm 722
log10() . . . . . Compute common logarithm 723
login. . . . . Log in or change user name. 723
logmsg . . . . . Hold COHERENT Login Message 724
long . . . . . Data type. 724
longjmp() . . . . . Return from a non-local goto 724
look . . . . . Find matching lines in a sorted file 725
lp . . . . . Line printer driver 725
lpd. . . . . Line printer spooler daemon 726
lpioctl.h. . . . . Definitions for line-printer I/O control 726
lpr . . . . . Send to line printer spooler 727
lpskip . . . . . Terminate/restart current line printer listing 727
ls . . . . . List directory's contents 728
lseek() . . . . . Set read/write position. 729
ltol3() . . . . . Convert long integer to file system block number 730
lvalue 730
m4. . . . . Macro processor. 732
machine.h . . . . . Machine-dependent definitions 734
macro 734
madd() . . . . . Add multiple-precision integers. 734

mail . . . . . Electronic mail system 735
mail . . . . . Computer mail 736
main() . . . . . Introduce program's main function 738
major number . . . . . Device numbering. 739
make . . . . . Program building discipline 739
malloc() . . . . . Allocate dynamic memory 742
malloc.h . . . . . Definitions for memory-allocation functions 744
man . . . . . Manual macro package 744
man . . . . . Print Lexicon entries 745
manifest constant. 746
math.h . . . . . Declare mathematics functions. 746
mathematics library 746
mboot . . . . . Master boot block for hard disk 747
mcmp() . . . . . Compare multiple-precision integers 748
mcopy() . . . . . Copy a multiple-precision integer 748
mdata.h . . . . . Define machine-specific magic numbers 748
mdiv() . . . . . Divide multiple-precision integers 748
me. . . . . MicroEMACS screen editor 749
mem. . . . . Physical memory file 756
memccpy() . . . . . Copy a region of memory up to a set character 756
memchr() . . . . . Search a region of memory for a character 757
memcmp() . . . . . Compare two regions 758
memcpy() . . . . . Copy one region of memory into another 759
memmove() . . . . . Copy region of memory into area it overlaps 759
memok() . . . . . Test if the arena is corrupted 760
memory allocation 760
memset() . . . . . Fill an area with a character 761
mesg . . . . . Permit/deny messages from other users 761
min() . . . . . Read multiple-precision integer from stdin 762
minit() . . . . . Condition global or auto multiple-precision integer 762
minor number . . . . . Device numbering. 762
mintfr() . . . . . Free a multiple-precision integer 762
misc . . . . . Library of miscellaneous functions. 763
mitom() . . . . . Reinitialize a multiple-precision integer 763
mkdir . . . . . Create a directory 763
mkfnames . . . . . Generate data base of user names 764
mkfs. . . . . Make a new file system. 764
mknod() . . . . . Create a special file 767
mknod . . . . . Make a special file or named pipe 768
mktemp() . . . . . Generate a temporary file name 768
mneg() . . . . . Negate multiple-precision integer. 768
mnttab.h . . . . . Structure for mount table 769
modem 769
modemcap . . . . . Modem-description language 773
modem control 775
modeminit . . . . . Initialize a modem 776
modf() . . . . . Separate integral part and fraction. 776
modulus 777
mon.h. . . . . Read profile output files 777
more . . . . . Display text one page at a time 777
motd . . . . . File that holds message of the day 780
mount() . . . . . Mount a file system. 780
mount.all . . . . . Mount file systems at boot time 781

mount. . . . . Mount a file system. 781
mount.h . . . . . Define the mount table. 782
mout(). . . . . Write multiple-precision integer to stdout 782
mprec.h. . . . . Multiple-precision arithmetic 782
ms . . . . . Manuscript macro package 782
MS-DOS . . . . . That other operating system 784
msg . . . . . Message device driver 789
msg . . . . . Send a brief message to other users 790
msg.h . . . . . Definitions for message facility 790
msgctl() . . . . . Message control operations 790
msgget(). . . . . Get message queue 792
msgrcv() . . . . . Receive a message 793
msgs . . . . . Read messages intended for all COHERENT users 794
msgsnd() . . . . . Send a message 795
msig.h . . . . . Machine-dependent signals 797
msqrt() . . . . . Compute square root of multiple-precision integer 797
msub() . . . . . Subtract multiple-precision integers. 797
mtab.h . . . . . Currently mounted file systems 797
mtioctl.h . . . . . Magnetic-tape I/O control 798
mtoi() . . . . . Convert multiple-precision integer to integer. 798
mtos() . . . . . Convert multiple-precision integer to string 798
mtype() . . . . . Return symbolic machine type 799
mtype.h. . . . . List processor code numbers 799
mult() . . . . . Multiply multiple-precision integers 799
multiple-precision mathematics 799
mv. . . . . Rename files or directories 803
mvfree() . . . . . Free multiple-precision integer 803
n.out.h . . . . . Define n.out file structure 804
named pipe 804
ncheck . . . . . Print file names corresponding to i-node 804
newgrp . . . . . Change to a new group. 805
newusr . . . . . Add new user to COHERENT system 805
nkb . . . . . Device driver for console keyboard 806
nlist() . . . . . Symbol table lookup 808
nm. . . . . Print a program's symbol table 809
notmem() . . . . . Check if memory is allocated 810
nptx . . . . . Generate permutations of users' full names 811
nroff. . . . . Text-formatting language 812
NUL 819
NULL 819
null . . . . . The "bit bucket". 820
nybble 820
object format. 821
od . . . . . Print an octal dump of a file. 821
open() . . . . . Open a file 821
operator 822
PAGER . . . . . Specify Output Filter 825
param.h . . . . . Define machine-specific parameters 825
passwd . . . . . Password file format 825
passwd . . . . . Set/change login password 826
paste . . . . . Merge lines of files 826
patch . . . . . Modify portions of an executable 827
PATH . . . . . Directories that hold executable files 829

path() . . . . . Path name for a file 829
path.h. . . . . Define/declare constants and functions used with path 830
paths . . . . . Routing data base for mail 830
pattern 831
pause() . . . . . Wait for signal 832
pax . . . . . Portable archive interchange 832
pclose() . . . . . Close a pipe 832
Permissions . . . . . Format of UUCP permissions file 833
perror() . . . . . System call error messages 835
phone . . . . . Print numbers and addresses from phone directory. 835
pipe() . . . . . Open a pipe 836
pipe 838
pnmatch() . . . . . Match string pattern 838
pointer 839
poll.h . . . . . Define structures/constants used with polling devices 842
popd. . . . . Pop an item from the directory stack 842
popen() . . . . . Open a pipe 842
port 842
portability 843
pow() . . . . . Raise multiple-precision integer to power 843
pow() . . . . . Compute a power of a number 844
pr . . . . . Paginate and print files 844
prep . . . . . Produce a word list 845
print. . . . . Echo text onto the standard output 845
printer 846
printf() . . . . . Print formatted text. 848
proc.h. . . . . Define structures/constants used with processes 851
process 851
prof . . . . . Print execution profile of a C program 851
profile . . . . . Set user's environment at login 851
prps . . . . . Prepare files for PostScript-compatible printer 852
ps . . . . . Print process status 854
PS1 . . . . . User's default prompt 855
PS2 . . . . . Prompt when user continues command onto additional lines . 855
ptrace() . . . . . Trace process execution 856
pun 857
pushd . . . . . Push an item onto the directory stack 857
putc() . . . . . Write character into stream 858
putchar() . . . . . Write a character onto the standard output 858
puts() . . . . . Write string onto standard output 859
putw() . . . . . Write word into stream. 859
pwd . . . . . Print the name of the current directory 860
pwd.h . . . . . Declare password structure 860
qfind . . . . . Find all files with a given name 861
qsort() . . . . . Sort arrays in memory 861
quot . . . . . Summarize file-system usage 862
ram . . . . . Driver for manipulating RAM 863
rand() . . . . . Generate pseudo-random numbers 864
random access 865
ranlib 865
ranlib . . . . . Create index for object library. 865
rc . . . . . Perform standard maintenance chores 866
read() . . . . . Read from a file 866

read . . . . . Assign values to shell variables 866
readonly . . . . . Storage class 867
readonly . . . . . Mark a shell variable as read only 867
read-only memory 868
realloc() . . . . . Reallocate dynamic memory 868
reboot . . . . . Reboot the COHERENT system 868
ref . . . . . Display a C function header 868
register . . . . . Storage class 869
register variable 869
rename . . . . . How to rename a file 869
restor . . . . . Restore file system 870
return . . . . . Return a value and control to calling function 872
rev . . . . . Print text backwards 872
rewind() . . . . . Reset file pointer 873
rindex() . . . . . Find a character in a string 873
rm . . . . . Remove files 874
rmdir . . . . . Remove directories 875
root 875
rpow() . . . . . Raise multiple-precision integer to power 875
RS-232 . . . . . COM port wiring 875
rvalue 877
sa . . . . . Print a summary of process accounting 878
sbrk() . . . . . Increase a program's data space 879
scanf() . . . . . Accept and format input 879
scat . . . . . Print text files one screenful at a time 881
sched.h . . . . . Define constants used with scheduling 883
SCSI . . . . . SCSI device drivers 883
sdiv() . . . . . Divide multiple-precision integers 883
SECONDS . . . . . Number of seconds since current shell started 884
security 884
sed . . . . . Stream editor 885
seg.h . . . . . Definitions used with segmentation 887
sem . . . . . Semaphore device driver 888
sem.h . . . . . Definitions used by semaphore facility 888
semctl() . . . . . Control semaphore operations 889
semget() . . . . . Get a set of semaphores 890
semop() . . . . . Perform semaphore operations 891
set . . . . . Set shell option flags and positional parameters 893
setbuf() . . . . . Set alternative stream buffers 895
setgid() . . . . . Set group id and user id 895
setgrent() . . . . . Rewind group file 896
setjmp() . . . . . Perform non-local goto 896
setjmp.h . . . . . Define setjmp() and longjmp() 897
setpwent() . . . . . Rewind password file 897
settz() . . . . . Set local time zone 898
setuid() . . . . . Set user id 898
sgtty . . . . . General terminal interface 898
sgtty.h . . . . . Definitions used to control terminal I/O 904
sh . . . . . The Bourne shell 904
SHELL . . . . . Name the default shell 914
shellsort() . . . . . Sort arrays in memory 914
shift . . . . . Shift positional parameters 915
shm . . . . . Shared memory device driver 915

shm.h . . . . . Definitions used with shared memory 916
shmctl() . . . . . Control shared-memory operations 917
shmget() . . . . . Get shared-memory segment 918
short . . . . . Data type 919
shutdown . . . . . Shut down the COHERENT system 919
signal() . . . . . Specify disposition of a signal 919
signal.h . . . . . Declare signals 921
signame . . . . . Array of names of signals 921
sin() . . . . . Calculate sine 921
sinh() . . . . . Calculate hyperbolic sine 921
size . . . . . Print size of an object file 922
sizeof . . . . . Return size of a data element 922
sleep() . . . . . Suspend execution for interval 923
sleep . . . . . Stop executing for a specified time 923
sload() . . . . . Load device driver 923
smult() . . . . . Multiply multiple-precision integers 924
sort . . . . . Sort lines of text 924
spell . . . . . Find spelling errors 925
split . . . . . Split a text file into smaller files 926
spow() . . . . . Raise multiple-precision integer to power 926
sprintf() . . . . . Format output 927
sqrt() . . . . . Compute square root 927
srand() . . . . . Seed random number generator 927
srcpath . . . . . Find source files 928
ss . . . . . Future Domain/Seagate SCSI device driver 928
sscanf() . . . . . Format a string 930
stack 931
standard error 931
standard input 931
standard output 932
stat() . . . . . Find file attributes 932
stat.h . . . . . Definitions and declarations used to obtain file status 934
static . . . . . Declare storage class 934
stdarg.h . . . . . Header for variable numbers of arguments 934
stddef.h . . . . . Header for standard definitions 935
stderr 935
stdin 935
STDIO 935
stdio.h . . . . . Declarations and definitions for I/O 936
stdlib.h . . . . . Declare/define general functions 936
stdout 937
sticky bit 937
stime() . . . . . Set the time 938
storage class 938
strcat() . . . . . Concatenate strings 938
strchr() . . . . . Find a character in a string 939
strcmp() . . . . . Compare two strings 939
strcoll() . . . . . Compare two strings, using locale-specific information 939
strcpy() . . . . . Copy one string into another 940
strcspn() . . . . . Return length a string excludes characters in another 940
stream 940
stream.h . . . . . Definitions for message facility 941
strerror() . . . . . Translate an error number into a string 941

string.h . . . . . Declarations for string library 941
string functions 942
strings . . . . . Print all character strings from a file 944
strip . . . . . Strip debug, relocation, and symbol tables from executable file 944
strlen() . . . . . Measure the length of a string 944
strncat() . . . . . Append one string onto another 945
strncmp() . . . . . Compare two strings 945
strncpy() . . . . . Copy one string into another 945
strpbrk() . . . . . Find first occurrence of a character from another string 947
strrchr() . . . . . Search for rightmost occurrence of a character in a string . . . 947
strspn() . . . . . Return length a string includes characters in another 948
strstr() . . . . . Find one string within another 948
strtod() . . . . . Convert string to floating-point number 949
strtok() . . . . . Break a string into tokens 949
strtok() . . . . . Break a string into tokens 950
strtol() . . . . . Convert string to long integer 951
strtoul() . . . . . Convert string to unsigned long integer 952
struct . . . . . Data type 954
structure 955
structure assignment 955
strxfrm() . . . . . Transform a string 955
stty() . . . . . Set terminal modes 956
stty . . . . . Set/print terminal modes 957
su . . . . . Substitute user id, become superuser 959
suload() . . . . . Unload device driver 959
sum . . . . . Print checksum of a file 959
superuser 960
swab() . . . . . Swap a pair of bytes 960
switch . . . . . Test a variable against a table 961
sync() . . . . . Flush system buffers 961
sync . . . . . Flush system buffers 962
system() . . . . . Pass a command to the shell for execution 962
system maintenance 962
tail . . . . . Print the end of a file 964
tan() . . . . . Calculate tangent 964
tanh() . . . . . Calculate hyperbolic cosine 964
tape . . . . . Magnetic tape devices 965
tar . . . . . V7 tape archive manager 966
technical information 968
tee . . . . . Branch pipe output 968
tempnam() . . . . . Generate a unique name for a temporary file 969
TERM . . . . . Name the default terminal type 969
termcap . . . . . Terminal-description language 969
terminal 976
terminal-independent operations 978
termio.h . . . . . Definitions used with terminal input and output 979
test . . . . . Evaluate conditional expression 979
tgetent() . . . . . Read termcap entry 981
tgetflag() . . . . . Get termcap Boolean entry 981
tgetnum() . . . . . Get termcap numeric feature 981
tgetstr() . . . . . Get termcap string entry 982
tgoto() . . . . . Read/interpret termcap cursor-addressing string 982
time() . . . . . Get current time 982

time 983
time . . . . . Time the execution of a command 983
time.h . . . . . Give time-description structure. 984
timeb.h . . . . . Declare timeb structure 984
timef.h . . . . . Definitions for user-level timed functions 984
timeout.h. . . . . Define the timer queue. 984
times() . . . . . Obtain process execution times. 984
times . . . . . Print total user and system times 985
times.h . . . . . Definitions used with times() system call 985
TIMEZONE. . . . . Time zone information 985
tmpnam(). . . . . Generate a unique name for a temporary file. 987
tolower() . . . . . Convert characters to lower case 987
touch . . . . . Update modification time of a file 988
toupper() . . . . . Convert characters to upper case 988
tputs(). . . . . Read/decode leading padding information 989
tr . . . . . Translate characters 989
trap . . . . . Execute command on receipt of signal 990
troff . . . . . Extended text-formatting language. 991
true . . . . . Unconditional success 996
tsort. . . . . Topological sort 996
tty . . . . . Print the user's terminal name 996
tty.h. . . . . Define flags used with tty processing 997
ttyname() . . . . . Identify a terminal 997
ttys . . . . . Describe terminal ports 997
ttyslot() . . . . . Return a terminal's line number 999
ttystat. . . . . Get terminal status. 999
type checking 1000
typedef . . . . . Define a new data type. 1000
type promotion 1000
types.h . . . . . Declare system-specific data types 1000
typeset . . . . . Set/list variables and their attributes 1001
typo . . . . . Detect possible typographical and spelling errors 1001
umask(). . . . . Set file-creation mask 1003
umask . . . . . Set the file-creation mask 1003
umount() . . . . . Unmount a file system 1004
umount. . . . . Unmount file system 1004
unalias . . . . . Remove an alias. 1004
uncompress . . . . . Uncompress a compressed file 1005
ungetc(). . . . . Return character to input stream 1005
union . . . . . Multiply declare a variable. 1005
uniq . . . . . Remove/count repeated lines in a sorted file 1006
units . . . . . Convert measurements 1006
unlink() . . . . . Remove a file. 1008
unmkfs . . . . . Construct a prototype file system 1008
unsigned . . . . . Data type. 1009
until. . . . . Execute commands repeatedly 1009
update . . . . . Update file systems periodically 1009
uproc.h . . . . . Definitions used with user processes 1010
USER . . . . . Name user's identifier 1010
ustar . . . . . Process tape archives. 1010
utime() . . . . . Change file access and modification times 1011
utmp.h . . . . . Login accounting information. 1012
utsname.h . . . . . Define utsname structure 1012

uucheck . . . . . Sanity check the UUCP system 1012
uucico . . . . . Transmit data to or from a remote site 1013
UUCP . . . . . Unattended communication with remote systems 1014
uucp . . . . . Ready files for transmission to other systems 1019
uucpname . . . . . Set the system's UUCP name 1020
uudecode . . . . . Decode a binary file sent from a remote system 1020
uuencode . . . . . Encode a binary file for transmission 1020
uuinstall . . . . . Install UUCP 1021
uulog . . . . . Examine UUCP operations 1022
uumvlog . . . . . Archive UUCP log files 1022
uuname . . . . . List UUCP names of known systems 1022
uurmlock . . . . . Remove UUCP lock files 1023
uutouch . . . . . Touch a file to trigger uucico poll 1023
uux . . . . . Execute a command on a remote system 1024
uuxqt . . . . . Execute commands requested by a remote system 1026
v7sgtty.h . . . . . UNIX Version 7-style terminal I/O 1027
va_arg() . . . . . Return pointer to next argument in argument list 1027
va_end() . . . . . Tidy up after traversal of argument list 1027
va_start() . . . . . Point to beginning of argument list 1028
variable arguments 1028
vi . . . . . Clone of Berkeley-style screen editor 1029
view . . . . . Screen-oriented viewing utility 1030
virec . . . . . Recover the modified version of a file after a crash 1030
void . . . . . Data type 1031
volatile . . . . . Qualify an identifier as frequently changing 1031
wait() . . . . . Await completion of a child process 1032
wait . . . . . Await completion of background process 1032
wall . . . . . Send a message to all logged-in users 1033
wc . . . . . Count words, lines, and characters in text files 1033
whence . . . . . List a command's type 1033
whereis . . . . . Locate source, binary, and manual files 1034
which . . . . . Locate executable files 1035
while . . . . . Introduce a loop 1035
while . . . . . Execute commands repeatedly 1035
who . . . . . Print who is logged in 1036
wildcards 1036
write() . . . . . Write to a file 1037
write . . . . . Converse with another user 1037
xgcd() . . . . . Extended greatest-common-divisor function 1039
yacc . . . . . Parser generator 1040
yes . . . . . Print infinitely many responses 1041
zcat . . . . . Concatenate a compressed file 1042
zerop() . . . . . Indicate if multi-precision integer is zero 1042
17. Appendix: Error Messages 1043
COHERENT System Error Messages 1043
Compiler Error Messages 1044
Assembler Error Messages 1045
cpp Error Messages 1045
cc0 Error Messages 1048
cc1 Error Messages 1057
cc2 Error Messages 1058
Linker Error Messages 1058
fsck Error Messages 1059

Initialization 1059
Phase 1: Check Blocks and Sizes 1060
Phase 1b: Rescan for more Duplicates 1060
Phase 2: Check Path Names 1060
Phase 3: Check Connectivity 1061
Phase 4: Check Reference Counts 1062
Phase 5: Check Free List 1062
Phase 6: Salvage Free List 1063
Cleanup 1063
General Messages 1063
make Error Messages 1064
nroff Error Messages 1065
Index 1069

Section 1:

Introduction

COHERENT is a professional operating system designed for use on the PC-AT and compatibles. It has many of the same features and functionality of the UNIX operating system, but is the creation of Mark Williams Company. COHERENT gives your computer multi-tasking, multi-user capabilities without the tremendous overhead, both in hardware and money, required by current editions of UNIX. COHERENT is what UNIX once was: an efficient system of selected tools and well-designed utilities, that brings out the best in modest computer systems.

The COHERENT system consists of the following:

- A fully multi-tasking, multi-user kernel and Bourne shell.
- The Mark Williams C compiler, a linker, an assembler, a preprocessor, and other tools.
- A suite of commands, including editors, languages, tools, and utilities.
- Drivers for peripheral devices, including terminals, ASCII printers, and the Hewlett-Packard LaserJet printer.
- Libraries, including the standard C library and the mathematics library.
- Sample programs, including full source code for the MicroEMACS editor.

Hardware Requirements

COHERENT runs on an IBM PC-AT or any totally compatible computer that has at least 640 kilobytes of RAM and at least one high-density floppy disk drive and a hard disk.

Before you begin to install COHERENT, be sure to check the release notes that accompany this manual for a list of tested hardware and known incompatibilities.

How To Use This Manual

This manual is in two parts. The first part consists of a set of tutorials that introduce COHERENT and its utilities.

If you are new to COHERENT, you should first read the following tutorial, *Using the COHERENT System*. This gives you an overview of COHERENT, and will get you up and running. It also includes information for advanced users on how to administer a COHERENT system properly.

The following tutorials introduce many of the COHERENT tools and utilities, including the editors MicroEMACS, **ed**, and **sed**; the C language; the language tools **awk**, **lex**, and **yacc**; **bc**, the multi-precision calculator; **make**, the COHERENT programming discipline; and many others.

The Lexicon

The second half of this manual is take up by the Lexicon. The Lexicon consists of approximately 1,000 articles that summarize all library routines, system calls, and commands available under the COHERENT system. It also includes numerous articles that define terminology and give technical information.

The articles are arranged in alphabetical order, to make it easy for you to find information on any topic. The articles are also linked via their cross-references into a tree structure, with the "root" of the tree being the article titled **Lexicon**. You can trace from any one article in the Lexicon to any other article simply by following the cross-references up and down the Lexicon's tree. The Index also references all topics discussed in the Lexicon or the tutorials, should you wish to look something up quickly.

If you are unfamiliar with a technical term used in this manual, look it up in the Lexicon. Chances are, you will find a full explanation. If you are not sure how to use the Lexicon, look up the entry for **Lexicon** within the Lexicon. This will help you get started. If you have struggled with multi-volume manuals for other operating systems, we think you will quickly come to appreciate the Lexicon.

User Registration and Reaction Report

Before you continue, fill out the User Registration Card that came with your copy of COHERENT. When you return this card, you become eligible for direct telephone support from the Mark Williams Company technical staff, and you will automatically receive information about all new releases and updates.

If you have comments or reactions to the COHERENT software or documentation, please fill out and mail the User Reaction Report included at the end of the manual. We especially wish to know if you found errors in this manual. Mark Williams Company needs your comments to continue to improve COHERENT.

Technical Support

Mark Williams Company provides free technical support to all registered users of COHERENT. If you are experiencing difficulties with COHERENT, outside the area of programming errors, feel free to contact the Mark Williams Technical Support Staff. You can telephone during business hours (Central time), send electronic mail, or write. This support is available *only* if you have returned your User Registration Card for COHERENT.

If you telephone Mark Williams Company, please have at hand your manual for COHERENT, as well as your serial number and version number. Please collect as much information as you can concerning your difficulty before you call. If you write, be sure to include the product serial number (from the sticker on the floppy disks) and your return address.

Section 2:

Using the COHERENT System

This tutorial introduces the COHERENT system. It introduces such basic concepts as *command* and *file system*, and walks you through simple exercises to help you gain some familiarity with the dimensions of COHERENT. If you are new to COHERENT, you should read through this tutorial first. Not every section in here will be immediately useful to every user; for example, a beginner will probably not need to study the section on system administration, at least at first. But sooner or later, you will need to work with all of the material in this tutorial.

If you are unfamiliar with what an *operating system* is, or if you are unsure how COHERENT differs from other operating systems (such as MS-DOS), turn to the Lexicon article for **COHERENT**. There, you will find a brief description of what an operating system is and what makes COHERENT special.

Before you can begin to use this tutorial, you must install COHERENT on your computer. If you have not yet done so, turn to the Release Notes that came with this manual and follow the directions in them.

How Do I Begin?

For everyone, there's that first time. You have installed COHERENT on your computer, you've checked the file system, mounted all of your file systems, and have gone into multi-user mode. Now you are sitting in front of your computer and all you see on your screen is the enigmatic phrase:

```
Coherent login:
```

"What," you ask yourself, "do I do now?" Well, the rest of this section will tell you how to get started with COHERENT.

Logging in

To begin, you must *log in*. Unlike MS-DOS, COHERENT is a multi-user system: many people can use the same computer, accessing it either via terminals that you plug into the computer's serial ports, or via modem. Each user owns his personal set of files, his special way of setting up his environment, his own mailbox, and other things which are special to him alone. Because many people can use COHERENT, before you begin to work with COHERENT you must tell it who you are. This process of identifying yourself to COHERENT is called *logging in*. That mysterious prompt

```
Coherent login:
```

is COHERENT's way of asking you who you are.

To log in, type your personal login identifier. You set this identifier when you first installed COHERENT on your computer. Most people set their login identifier to their initials or their first names, usually in all lower case letters. Once you type your login identifier, press the **<Return>** key (sometimes labelled as **<Enter>**). If you did not set up a login for yourself during installation, log in as the superuser **root** and add one for yourself. For information on how to log in as the superuser, see below. For information on how to add a new user,

see the section on **Adding a New User**, below, or see the Lexicon article for the command **newusr**.

While you were installing COHERENT on your system, you were given the option of setting a password for your login identifier. This is done to stop other users from logging in as yourself — or to keep outside "crackers" from dialing into your system and vandalizing it. If you did set a password, after you enter your login identifier COHERENT will prompt you for it with the following prompt:

```
Password:
```

Type your password. Note that COHERENT does *not* display the password on the screen as you type it; this is to prevent bystanders from seeing your password over your shoulder as you enter it. After you type your password, again type **<Return>**.

If you entered your login identifier and passwords correctly, COHERENT will display the command prompt:

```
$
```

This is COHERENT's way of saying, "Give me a command, I'm ready to go!" If you made a mistake while logging in, either with the identifier or the password, COHERENT will reply,

```
Sorry!
```

and display its

```
Coherent login:
```

prompt again. Try again, until you do manage to log in. If you have received the '$', congratulations! COHERENT is now ready to work with you.

Special Terminal Keys

The next sections will introduce you to a few elementary COHERENT commands. Before we continue, however, you must first become familiar with a few special keys on your computer's keyboard, and with the special meanings they have to the COHERENT system.

One special key on the keyboard will be used frequently in your work: the **<Return>** key. As noted above, this key is sometimes labelled **<Enter>**.

You must conclude every command you type into COHERENT by pressing the **<Return>** key. This tells COHERENT that you have finished typing, and that you now want it to execute your command. COHERENT will not execute your command until you press this key.

Another special key is the **control** key. This key is usually labelled **Ctrl** or **cntl** or **cont**. Most terminals place it to the left of the keyboard. This key is used to send certain special characters.

The **ctrl** key is like another kind of shift key: to use it, hold it down while you press another key. For example, to send the computer a **<ctrl-D>** character, hold down the **ctrl** key, strike the **D** key, then release both keys.

Because control characters have no corresponding printable characters, in this tutorial they will be represented in the form:

```
<ctrl-D>
```

for the character **ctrl-D**.

While you are typing information into the COHERENT system, you can correct the information before it is processed. Two keys will help you do this. The first is the **<kill>** character, which erases the line entirely and allows you to begin again. This is usually **<ctrl-U>**.

The other key is the **<erase>** character, normally **<ctrl-H>** or **<backspace>**. This moves the cursor one character to the left, to erase the most recently typed character. **<ctrl-H>** also serves as the backspace key.

One more special key is the **<interrupt>** key. This key aborts a command before it normally finishes. By default, **<ctrl-C>** is the abort key on your keyboard.

Try Some COHERENT Commands

Now that you've logged in to your COHERENT system, try a few simple COHERENT commands to get a feel for COHERENT. Type the following examples just as they are shown, and observe what COHERENT does in response to each. Be sure to end each line with a **<Return>** .

The first example uses the command **cat**, to let you type a small chunk of text and save it in a file.

```
cat >file01
This is a sample COHERENT file.
<ctrl-D>
```

Remember, don't type **<ctrl-D>** literally — rather, hold down the **ctrl** key and press 'D' at the same time.

In the above script, the characters **cat** tell COHERENT to invoke its concatenation program. The characters **>file01** tells COHERENT to write what you type into a file that you name **file01**. The line

```
This is a sample COHERENT file.
```

is the text that COHERENT writes into **file01**. Finally, **<ctrl-D>** signals COHERENT that you have finished typing.

Now type:

```
cat file01
```

This command again invokes the concatenation program **cat**, but this time tell it to print on your screen the contents of **file01**, which you just created. In reply to your command, COHERENT should print on your screen:

```
This is a sample COHERENT file.
```

which is the text you entered in the previous exercise.

Finally, type the command:

```
lc
```

This command lists all of the files that you have in the current directory. In reply to your command, COHERENT should print on your screen:

```
Files:
    file01
```

which is the file you just created. (You may see other files as well.)

Congratulations! You have just made COHERENT work for you.

To review: The first command, **cat**, created a file and filled it with some text; the second **cat** typed the file out on your terminal; and the command **lc** printed the name of each of your files. The following sections of this tutorial describe each of these commands in more depth. Each command also has its own entry in the Lexicon, which appears in the second half of this manual; look there for a full description of each command, what it does, and how you can use it.

Giving Commands to COHERENT

Once you have logged into COHERENT, all of its resources are yours to command. COHERENT's *commands* give you control over these resources.

Every COHERENT command has the same structure: the *command name*, which tells COHERENT the command you want it to execute; and the *arguments*, which detail what you want the command to do, how you want it to do it, and to what you want it done.

Some commands consist only of the command name, and do not take arguments. For example, the command

```
lc
```

which was introduced in the previous section, has **lc** as the first part and prints the names of all files in the current directory, in columns. If you have no files, **lc** prints nothing.

The second part of the command consists of the *arguments* given to the command. (These are also known by the term *parameters*.) Arguments are separated from each other by spaces or tab characters.

The arguments of the command are further divided into *options* and *names*. *Names* usually name files; *options* modify the action of the command. An option is usually prefixed by a hyphen '-'.

An example of a *name* argument is shown in this example of a **cat** command:

```
cat file01
```

This command types the contents of **file01** on your terminal. The name argument is **file01**.

For an example of options, consider the command **ls**. **ls** lists your file names one name per line. Thus, typing

```
ls
```

produces a list of the form:

```
file01
```

However, **ls** can tell you more about a file than just its name. To see additional information about each file, type:

```
ls -l
```

The '-l' option to **ls** prints a "long" output, of the following form:

```
-rw-r--r-- 1 you      17 Sat Aug 15 17:20 file01
```

This listing shows the size of the file, the date it was created or last modified, and its degree of protection. The letters to the left of the listing give the permissions for the file; these describe who is allowed to do what to the file. These are described in detail in the Lexicon

articles for the commands **ls** and **chmod**. The other entries on that line respectively name the owner of the file (in this case, *you*); the size of the file in bytes; the date and time the file was last modified; and finally, the file's name.

As an example of combining an option parameter with a name parameter, consider the command:

```
ls -l file01
```

This invokes the command **ls**, tells it to print a long listing, and tells it to list only the file **file01**.

As you will see in the following sections, almost all COHERENT commands have this syntax.

help, man: Help with Commands

The COHERENT system has two commands that give information about other commands: the **help** command, which prints a brief summary of how to use a command; and the **man** command, which prints the full Lexicon entry for that command on your screen.

To find out about the **help** command, type

```
help
```

by itself, or type:

```
help help
```

The latter command tells **help** to print the help entry for the **help** command itself.

To get information on the **lc** command, type:

```
help lc
```

To obtain detailed information on a command, use the **man** command. (**man** is short for "manual".)As noted above, the **man** prints on your screen a duplicate of that command's entry in the Lexicon. To learn more about the **man** command itself, type:

```
man man
```

If your screen fills with information, **man** will wait for you to type **<Return>** to continue. This is to prevent you from missing information should it scroll too fast. **man** also waits for you to type **<Return>** after it prints the last line of the description.

Our survey of elementary commands will conclude by describing two important tasks: how to reboot the computer, and how to log out.

Shutting Down COHERENT and Rebooting

Under many operating systems, such as MS-DOS, rebooting is as simple as pressing a couple of keys or cycling power on the computer. The COHERENT system, however, is a multi-user, multi-tasking operating system that is more sophisticated than MS-DOS or similar operating systems. COHERENT maintains an elaborate system of internal buffers that are designed to reduce the frequency with which a program has to read data from, or write data to, the hard disk. If you were just to turn the computer off and turn it on again, all of the data in those buffers would be lost. At the very least, each user would lose whatever data he was working with at the time; at worst, the COHERENT file system could be damaged and files lost.

For this reason, it is extremely important that you shut down COHERENT properly. You *must* follow these procedures if you want to shut off the computer, or if you wish to reboot MS-DOS.

To shut down COHERENT, do the following:

- Log in as the superuser **root** by typing the following command:

  ```
  su root
  ```

  COHERENT will ask you for the superuser's password; type the password that you assigned to the superuser when you installed COHERENT on your computer. The Lexicon article on **superuser** describes what the superuser is; as will later sections of this tutorial.

- Once you have logged in as the superuser, type the following command:

  ```
  /etc/shutdown
  ```

  As its name implies, this command shuts down the COHERENT system. The command will ask you if you really, truly wish to shut down COHERENT; reply 'y', for "yes".

- COHERENT will indicate that it has returned to *single-user mode* by printing the prompt '#'. When this prompt appears, type the command:

  ```
  sync
  ```

  This command flushes all buffers and writes their contents to the hard disk. When you first type this command, you should hear or see the disk in action. Now, type it again. You probably will not hear any activity from the disk: that is because the buffers have been flushed and nothing remains to be written to the disk.

- Now, you can turn the computer off. If you wish to reboot COHERENT, instead of turning the computer off type the command:

  ```
  /etc/reboot
  ```

  This will reboot COHERENT automatically. Or, you can type **<ctrl><alt><del>**, or press the reset button on your computer (should it have one).

  After you have rebooted your computer, just sit back and wait until you receive the **Coherent login:** prompt on your screen.

  If you wish to reboot MS-DOS, type the command:

  ```
  /etc/reboot
  ```

  Instead of sitting back, however, watch the computer: wait until you see the computer attempting to read from the floppy-disk drive. At that moment, press the number key that corresponds to the hard-disk sector on which you stored MS-DOS, from 0 to 7. For example, if MS-DOS is kept on partition 2, then press **2** when the computer is attempting to read the floppy-disk drive. Be sure to press the number key that is on the main bank of keys, — *not* the key on the numeric keypad.

That's all there is to it. Shutting down is relatively simple and straightforward; but if you do not take the time to shut COHERENT down properly, you will find that you have destroyed some or all of your data.

Logging Out

As noted above, *logging in* tells COHERENT who you are and that you wish to work with COHERENT for a while. When you have finished working with COHERENT, you must tell COHERENT that you are done for now. This process is called *logging out*.

There are three ways to log out. Each involves typing a special command to the COHERENT prompt. The first way is to type **<ctrl-D>** at the COHERENT prompt. The second is to type the command:

```
login
```

which logs you out and prepares for another login.

The third way is to type the command:

```
exit
```

Each of these commands has the same effect: the COHERENT system flushes all buffers that you "own" and prints the prompt

```
Coherent login:
```

on your screen. At this point, you cannot issue any commands to COHERENT; but you (or someone else) can log into COHERENT from this terminal.

Please note that logging out is *not* the same as shutting down COHERENT. When you shut down COHERENT, you are shutting down the entire system. When you log out, however, you are simply ceasing to work with COHERENT. After you log out, COHERENT continues to work on its own: organizing files, exchanging information with other computers via modem, executing programs for users who have logged in via modem or other terminals, and in general making itself useful. If you shut off the computer after you log out, you will damage the file system, just the same as if you shut it off while you were logged in.

The following sections in this tutorial will go into COHERENT's commands in much more detail. All, however, will build on the elementary actions presented here: logging into COHERENT; issuing commands; receiving responses from COHERENT; and logging out.

Working With Files and Directories

The *file* and the *directory* are the cornerstones of the COHERENT system. Practically everything you do on the system will involve files: changing files, invoking files, transmitting or receiving files, filling files up or emptying files out. And directories let you organize masses of files into a rational hierarchy.

This section discusses manipulating files and directories under the COHERENT system. It covers the following:

- What *file* and *directory* mean to COHERENT
- Introduces the commands for manipulating files, directories and their contents
- Discusses more advanced topics, such as creating and mounting new file systems
- Tours the COHERENT file system

This section of the tutorial covers much ground in a relatively brief space. Readers who are new to personal computers should concentrate on the earlier sub-sections, which cover elementary topics; whereas more experienced readers may wish to concentrate on the later sub-sections, which cover the more technical material.

File Names

A *file* is a mass of electronic impulses that is given a name and stored on a disk. Files are given names to make them easy for you to retrieve. COHERENT has rules about how files can be named, to ensure that each file's name is unique.

The following are examples of legal file names:

```
.profile
File01
cmd.sh
file01
test.c
```

File names are generally made up of upper-case and lower-case letters and numbers. COHERENT, unlike MS-DOS, distinguishes capital letters from lower-case letters; therefore, to COHERENT the file names **File01** and **file01** are different.

Any character can be used to name a file, including a control character. We recommend, however, that you names files using only upper- or lower-case alphabetic characters, numerals, and the punctuation marks '.' or '_'.

The file name must not be more than 14 characters long. If you specify a longer name, characters beyond the 14th will be lopped off and thrown away. For example, COHERENT regards the file names

```
this_is_very_long_file_name_1
```

and

```
this_is_very_long_file_name_2
```

as being identical.

Introduction to Directories

A *directory* is a group of files that have been given a name. Directories let you organize files systematically. This may not seem important now, but as you work with COHERENT you will find that you accumulate hundreds, or even thousands, of files; without system of directories to organize files, you would quickly lose track of what each file held, and find it nearly impossible to find any given file within your system.

Because files are stored within directories, the complete name of a file actually consists of its name plus the name of the directory in which it is stored. This lets COHERENT distinguish files that have the same name but are stored in different directories. COHERENT uses the slash character '/' to distinguish a directory name from a file name; for example, to view the contents of file **junk** in directory **text_files**, you would use the command:

```
cat text_files/junk
```

This system of naming will be described in full in the next sub-section; for the moment, just bear in mind that for COHERENT to find a file, you must tell COHERENT not only the name of the file, but the name of the directory in which it is kept.

When you work with COHERENT, you are always "in" a directory. The directory you happen to be "in" at any given moment is called the *current directory*. The current directory is the one whose files you are working with at this moment. When you type the name of a

file and do not mention what directory it is stored in, COHERENT assumes that the file is kept in the current directory. COHERENT includes commands that let you shift from one directory to another.

When you log into COHERENT, COHERENT places you "in" a directory that you "own". This directory is called your *home directory*. You control all of the files in your home directory; it is your "base of operations" for working within COHERENT.

Path Names

As you may have deduced by now, a directory can contain both files and other directories. The directories within a directory may themselves contain both files and directories; which then may contain other files and directories; and so on.

This design of directories branching into other directories, which in turn branch into still other directories, is called *tree structured*. As the tree-metaphor implies, the COHERENT system of directories has a *root directory*, that is, a directory that is not contain in any other directory but from which all other directories descend, directly or indirectly. The name of the root directory is simply:

```
/
```

One subdirectory of the root directory is called **usr**. This subdirectory contains the home directories of all users. Other common paths for home directories are **/u** and **/usr/acct**. To list the names of all user directories, type the command:

```
lc /usr
```

If your login name is **henry**, then the command

```
lc /usr/henry
```

lists the names of the files in your home directory. Please note that in the argument **/usr/henry**, the first slash names the root directory; all subsequent slashes serve simply to separate one directory name from the next.

The name **/usr/henry** is called a *path name*. The term "path name" means the full name of a given file or directory — including all the directories that lead from the root directory to it.

Path names may be full or partial. All full path names begin with / for root, and continue with further subdirectory names. Path names that do not begin with a slash are partial; COHERENT automatically prefixes them with the path name of the current directory to make them complete before it uses them.

The elements of path names are separated by slashes, so if there were a file in **newdirectory** named **newfile**, you would refer to it as

```
newdirectory/newfile
```

The absence of a beginning slash indicates that the path name begins in the current directory. Thus, if your home directory name is **henry**, then another way to name the path to **newfile** is to type:

```
/usr/henry/newdirectory/newfile
```

The following diagram gives a rough description of the structure of the COHERENT file system:

```
              /
     ==================
   bin               usr
            ==================
          henry            other
```

Please note that unlike a real tree, the root of a tree structure has its root at the top rather than at the bottom. Here, the root directory '/' is at the top of the structure. It contains the directories **bin** and **usr** (among many others). Directory **usr** contains directories **henry** and **other** (again, among many others. These directories can contain many other directories and subdirectories.

In summary, a path name lists all the subdirectories leading from the root directory to the file in question. In the above example, **newfile** is a file in subdirectory **newdirectory**, which in turn is a file in the home directory **henry**, which is further a file in the directory **usr**. The directory **usr** is a file in the master or **root** directory for the system.

You don't need to specify all of this, fortunately, whenever you want to specify a file in a subdirectory. COHERENT assumes that partially specified path names are within the current directory. Therefore, you can specify a subdirectory by specifying the name of the directory first, followed by the rest of the path name.

COHERENT also allows two special abbreviations for directories. The abbreviation '..' always represents the current directory's *parent* directory. In the case of the directory **/usr/henry**, directory **usr** is the parent of directory **henry**. In other words, '..' stands for the directory in which the current directory resides. Every directory in the system except the root directory has a parent. For the root directory, '..' refers to itself.

Another directory abbreviation is '.', which means the *current* directory.

The following sub-sections describe the commands that COHERENT includes for manipulating files and directories. As you work with COHERENT, you will use these commands continually, so it would be worth your while to spend a little time learning them.

ls, lc: Listing Your Directory

This sub-section introduces two of the more commonly used commands: **ls** and **lc**. Both **ls** and **lc** list the files in a directory.

To see how these commands work, presume that your directory has the files created in previous sections and that you did not remove directory **newdirectory**. To list the files in your directory, simply use the command with no parameters:

```
ls
```

This produces a list of files, such as:

```
another
backup
doc1
doc2
file01
file02
newdirectory
stuff
```

The command **lc** also lists file names, but it prints the files and directories separately, in

columns across the screen. For example, typing

```
lc
```

gives something of the form:

```
Directories:
   backup newdirectory
Files:
   another doc1 doc2 file01 file02
   stuff
```

If you want to list files in a directory other than your own, name that directory as an argument to the command. For example, **/bin** is a directory in the COHERENT system that contains commands. Type

```
lc /bin
```

and **lc** will print the contents of **/bin**.

Both **ls** and **lc** can take options. An option is indicated by a hyphen '-'. The option must appear before any other argument. For example, to list only the files in the directory for user **carol**, leaving out any directories, use the **f** option with **lc**:

```
lc -f /usr/carol
```

Or, if you type the command

```
lc -f
```

the COHERENT system prints all of the files in the current directory. The following gives the commonly used options to the command **lc**:

-d	List directories only, omitting files
-f	List files only, omitting directories
-1	List files in single column format

ls produces a list of file names, one per line, and optionally much more information. To produce all the information, use the **-l** option (note that this is an "el", not a numeral 1):

```
ls -l
```

The following gives a sample of the long list that this option produces. Headings have been added to show the meaning of each column:

Mode	*#*	*Owner*	*Size, Bytes*	*Modification Date*	*Time*	*Name*
-rw-r--r--	1	you	17	Wed Aug 19	17:51	file01
drwxrwxrwx	2	you	32	Wed Aug 19	17:53	backup
-rw-r--r--	1	you	17	Wed Aug 19	17:53	doc1

The meaning of each column will be explained later. For now, note that the last column gives the name of each file, and the fourth column from the left gives the size of each file, in bytes.

cat: Print Contents of a File

The command **cat** opens and prints the contents of a text file — that is, a file of source code, a document, or a message file. For example, to list the contents of file **file01**, type:

```
cat file01
```

This command types the file's contents on the terminal (sometimes also called the *standard output*).

Another use for **cat** — the use from which it gets its name — is to *concatenate* several files on the standard output. For example, the command

```
cat one two three
```

prints the files **one**, **two**, and **three**, one after the other, on the screen.

You can use **cat** to concatenate several files into one file by *redirecting* the standard output into a file. The special character '>' tells COHERENT to redirect the standard output into a file. For example, the command

```
cat one two three >four
```

concatenates files **one two three** into file **four**. **four** need not exist prior to this command; if it does, its previous contents are replaced with the data redirected into it.

Redirection is a very useful feature of COHERENT that will be used through the rest of this tutorial. The '>' operator also gives an example of the set of operators that can be used with COHERENT commands. These operators, which increase the power of each COHERENT command, will be described in detail later in this tutorial.

more: List Files on the Screen

If the file you list with **cat** is more than 24 lines long, the beginning lines of the file scroll off the screen too quickly for you to read them. To ensure that you see all of the lines in the file, use the command **more**.

more prints a file in 24-line chunks. After it has listed a chunk of text, it pauses and waits for you to press **<space>**. If you call **more** with an option of **-s**,

```
more -s file
```

it will skip all blank lines that are in the text file.

mkdir: Create a Directory

The command **mkdir** creates a new directory. For example, to create a new directory named **newdirectory**, type the following command:

```
mkdir newdirectory
```

If you follow this command with **lc**, it lists your regular files, but it also lists **newdirectory** separately as a directory:

```
Directories:
    newdirectory
Files:
    file01    file02
```

To refer to any files in **newdirectory**, use its name in specifying the path name.

Now, create a file in the new directory:

```
cat >newdirectory/newfile
lines to be
contained in newfile
<ctrl-D>
```

This command copies lines to the file described by the partial path name **newdirectory/newfile**.

cd: Change Directory

The command **cd** changes the current working directory. For example, the command

```
cd newdirectory
```

moves you into directory **newdirectory** that you created in the previous sub-section. Now, if you type the command **lc**, to show the contents of the current directory, it will show the following:

```
Files:
    newfile
```

To return to the previous directory, use the command:

```
cd ..
```

As noted earlier, the abbreviation '..' always indicates the current directory's parent directory.

pwd: Print Working Directory

The command **pwd** prints the name of the current, or *working*, directory. For example, if your login name is **henry**, then if you type

```
pwd
```

you will see:

```
/usr/henry
```

Now, use the **cd** command to switch to directory **newdirectory**, as follows:

```
cd newdirectory
```

When you type

```
pwd
```

you will see:

```
/usr/henry/newdirectory
```

Finally, use the **cd** command to return to the previous directory, as follows:

```
cd ..
```

When you type

```
pwd
```

you now see:

```
/usr/henry
```

If you are ever unsure what directory you are in, use the **pwd** command.

mv, cp: Move and Copy Files

The command **mv** moves files. You can move a file from one name to another within the current directory (in effect rename the file), or you can move a file from one directory to another. The **mv** command takes two parameters: the first names the file to be moved; the second names either the new name that you are giving to the file, or the directory into which you are moving the file.

For example, to move file **file01** into directory **newdirectory**, type:

```
mv file01 newdirectory
```

To see where **file01** is now, type the following command:

```
lc newdirectory
```

The result is:

```
Files:
    newfile
```

To move **newfile** back into the current directory, use the command:

```
mv newdirectory/newfile
```

Remember, the abbreviation '.' always stands for the current directory.

As noted above, the **mv** command can also be used to rename files within the current directory. For example, to change the name of **newfile** to **oldfile**, use the following command:

```
mv newfile oldfile
```

If the current directory already has a file named **oldfile**, it will be thrown away and replaced with the file that used to be named **newfile**.

The command **cp** copies a file. This command has two parameters: the first names the file to be copied, and the second names the file or directory into which it is to be copied. For example, to copy **oldfile** in the current directory back into **newfile**, use the following command:

```
cp oldfile newfile
```

If **newfile** already exists, it will be replaced by a copy of **oldfile**.

If you wished to copy **newfile** into directory **newdirectory**, use the command:

```
cp newfile newdirectory
```

Now, when you type the command

```
lc newdirectory
```

you will see:

```
Files:
    newfile
```

As you can see, **newfile** has been copied into **newdirectory**. If **newdirectory** had already contained a file called **newfile**, that file would have been replaced with the newer **newfile** being copied into **newdirectory**.

The following example summarizes what's been presented so far about files and directories. For purposes of the example, assume that your login name is **henry**, and that you have in your home directory files **doc1** and **doc2** that you wish to back up for safekeeping.

Before you can back up these files, you must first create them. First, use the command **cat** to create file **file01**, as follows:

```
cat >doc1
a few
lines of
text
<ctrl-D>
```

Likewise, create file **doc2**:

```
cat >doc2
second file
with some text
<ctrl-D>
```

(Don't forget that **<ctrl-D>** means to hold the control key down and simultaneously type **D**.)

The command **lc** will now show you the files and directories in your current directory:

```
Directories:
    newdirectory
Files:
    doc1    doc2    newfile    oldfile
```

The next step is to create the directory to hold the back-up copies. To help remind yourself what the directory is for, name it **backup**.

```
mkdir backup
```

Now, **lc** shows you:

```
Directories:
    backup    newdirectory
Files:
    doc1    doc2    newfile    oldfile
```

The next step is to use **cp** to copy your files into **backup**:

```
cp doc1 backup
cp doc2 backup
```

After you issue these commands, **lc** still says:

```
Directories:
    backup     newdirectory
Files:
    doc1     doc2     newfile     oldfile
```

However, if you list the contents of subdirectory **backup**

```
lc backup
```

you will see:

```
Files:
    doc1 doc2
```

The files have been successfully copied into the back-up directory.

For a full description of these commands and the options available with each, see their respective entries in the Lexicon.

rm, rmdir: Remove Files and Directories

The command **rm** removes a file. For example, if you wish to remove file **doc2** in directory **backup**, type the following command:

```
rm backup/doc2
```

After typing this command, use the command **lc** to show the contents of directory **backup**, as follows:

```
lc backup
```

You should see:

```
Files:
    doc1
```

As you can see, file **doc2** has been removed.

You can remove several files at once, simply by listing them on the **rm** command's command line. For example:

```
rm file01 file02
```

removes files **file01** and **file02**.

Note that once you remove a file with **rm**, it is gone forever. The COHERENT system does not warn you if you **rm** several files at once; it will assume that you know what you're doing and carry out your command silently. For this reason, be careful when you use the **rm** command, or you may receive a rude surprise.

You cannot use the command **rm** to remove a directory. COHERENT does this to help prevent you from wiping out an entire file system with one simple **rm** command. To remove a directory, use the command **rmdir**. For example, to remove the directory **newdirectory**, type:

```
rmdir newdirectory
```

Note that before you can delete a directory, that directory must not have any files or directories in it. If you try to remove a directory that has files or directories in it, COHERENT will print an error message on your screen and refuse to remove the directory.

For a full description of these commands and the options available with each, see their respective entries in the Lexicon.

du, df: How Much Space?

Files occupy space on your hard disk. (A corollary to Parkinson's law states that files expand to fill the disk allotted to them.) It is somewhat disconcerting to attempt to save a large file, only to find that you have run out of disk space. To help you manage your hard disk, COHERENT includes the commands **du** and **df**.

The disk-usage command **du** tells you how much disk space the files in the current directory occupy. If the directory has sub-directories, these are listed separately. **du** prints disk usage in blocks; each block is 512 bytes (half a kilobyte).

The disk-free command **df** tells you how many blocks are left free on your disk. By default it prints information only about the file system you are now in.

If you find that you are running low on disk space, you must free up some space. You can do that by removing files you no longer need; by *compressing* files that you do not use often; or by backing files up to floppy disk and then removing them. We have already described how to remove files. Look in the Lexicon entry for the command **compress** for information on how to compress and uncompress files. Following sections in this tutorial will describe how to copy files to floppy disk.

For more information on these commands, see their respective entries in the Lexicon.

ln: Link Files

COHERENT has a feature that allows a file to have more than one name. When you create a file, you give it a name; COHERENT *links* the name you give the file with its internal system of managing files. (For more information on how COHERENT identifies files, see the Lexicon entry for **i-node**.) COHERENT allows you to give a file more than one name; another way of expressing this is to say that you can give a file *multiple links*..

To create a new link to an existing file, use the command **ln**. This command takes two arguments: the first names the file to which you wish to give a new link, and the second gives the name that you wish to link to that file. If the name you are linking to a file is already being used by a file, that name is unlinked from its current file and linked to the file named in the **ln** command line.

For example to link the file **doc1** to the name **another**, use the following command:

```
ln doc1 another
```

The "new" file has the same data in it as the "old" file; in fact, the names **doc1** and **another** are synonyms for the same file.

The next point is somewhat subtle. When you use the command **rm** to remove a file, what you are actually doing is breaking the link between that file and its name. The file is not actually removed from disk until all links are broken between it and all of its names. In the above example, if you use the command

```
rm another
```

to remove the file **another**, the file **doc1** remains in existence, and the data to which the names **another** and **doc1** remains on the disk. If you then use the command

```
rm doc1
```

to remove **doc1**, then you will have broken all links between that file and the COHERENT system, and COHERENT will remove it from the disk.

Links are useful if you wish a file to be used in two different contexts but have the same data. For example, if you file **doc1** in two different manuscripts, you can create links to the file in two different directories, one for each manuscript. Thus, any changes you make to the file under either its names will appear automatically in both manuscripts. Note that if you copy over one link to a file, all links still point to the same file. However, if you use either a command of the form

```
ln -f file01 file02
```

or a command of the form

```
mv file01 file02
```

only the link which is overwritten points to the new file; other links continue to point to the old file.

As always, see the Lexicon for a full description of the **ln** command.

File Permissions

As you recall, the command **ls -l** prints a mass of information about each file. The following repeats the information that appeared when you typed **ls -l**:

Mode	#	*Owner*	*Size, Bytes*	*Modification Date*	*Time*	*Name*
-rw-r--r--	1	you	17	Wed Aug 19	17:51	file01
drwxrwxrwx	2	you	32	Wed Aug 19	17:53	backup
-rw-r--r--	1	you	17	Wed Aug 19	17:53	doc1

Column 3 names the owner; in this example, **you** represents your login name, whatever you have set it to. Column 4 gives the size of the file, in bytes. Columns 5 through 7 give the day of the week and the date on which the file was last modified. Column 8 gives the time the file was last modified or, if the file was last modified more than a year ago, the year it was last modified. Column 9 gives the name of the file.

Column 1 gives the *mode* of the file. The mode summarizes the *permissions* attached to this file.

Before going further, the concept of file permissions should be reviewed. COHERENT is a multi-user operating system, which means that more than one person can log into the system, walk through its file system, execute commands, and manipulate files. Every user has files that she "owns" — that is, that she has created and that she wishes to protect against being altered or removed by others. After all, it would be disconcerting if you were to log into your system, only to find that some of your key files had been trashed by another user, without your knowledge or permission.

The COHERENT system protects files by its system of file permissions. Permissions have two aspects: the *type* of permission, and the *scope* of permission. There are three *types* of permission:

read permission
Permission to read a file.

write permission
Permission to write into a file.

execute permission
Permission to execute a file, assuming that file contains executable code instead of text.

Likewise, there are also three types of *scope:*

user The permissions extended to the owner of the file.

group The permissions extended to the group of users to which the owner belongs. For more information on what *group* is, see the Lexicon entry for **group**.

other The permissions extended to all other users.

The *mode* column describes all permissions attached to a file. It also gives other information about a file, such as whether the file is a directory. Taking the entry for file **file01** as an example, we see:

```
1  2   3   4  # Owner        Size   Date      Time   File name
-rw-r--r--  1 you            17 Sat Aug 15 17:20  file01
```

As you can see, the mode field is divided into four subfields, in this example labelled '1' through '4'.

Subfield 1 indicates whether this file is a directory. If the file were a directory, this would contain a **d**; otherwise, it contains a hyphen.

Subfields 2 through 4 describe the type of permission extended to, respectively, the owner, the owner's group, and other users. Each subfield consists of three characters. The first character indicates whether the file is readable; if it is, then the character is an 'r'; otherwise, it's a hyphen. The second character indicates whether the file is writable; if it is, then the character is a 'w'; otherwise, it's a hyphen. The third character indicates whether the file is executable; if it is, then the character is an 'x'; otherwise, it's a hyphen.

In the above example, file **file01** grants read and write permission to its owner, read permission to the other members of the owner's group, and read permission to all other users.

The COHERENT system has a set of default permissions that it applies to every file when it's created. To change this default set of permissions, use the command **umask**. For information about this command, see its entry in the Lexicon. To change the permissions of an existing file, use the command **chmod**, as described in the following sub-section.

chmod: Change File Permissions

To change the mode of a file, use the change-mode command **chmod**. For example, to protect file **doc1** in directory **backup** from being overwritten, use the command:

```
chmod -w backup/doc1
```

where the **-w** means "remove write permission" and is followed by the file name. Henceforth, if you try to write into this file, the COHERENT system will refuse to do so and will print an error message on your screen.

To allow other users to read the backup file **doc2**, type:

```
chmod o+r backup/doc2
```

where the letter **o** signifies "other users", and the **+r** tells **chmod** to grant read permission.

To see the new set of permissions, type the command:

```
ls -l backup
```

As you can see, the mode string has changed from what it was above.

Directory access permissions are similar to file access permissions in that they can easily be changed via command **chmod**. However, the permission bits have different meanings for directories. Permitting reads on a directory allows the user to see the contents of the directory via commands such as **lc** or **ls**; permitting execution on a directory allows access to the files in the directory; and permitting writes on a directory allows the user to create or delete files in the directory, regardless of the permissions on the actual file. The latter causes the most difficulty for new users since they mistakenly associate file deletion permissions with the actual file rather than with the directory containing the file.

Creating and Mounting a File System

Earlier, we described how the COHERENT system consists of a tree of directories; and how that tree branches from the root directory '/'. This is a useful description, and true as far as it goes; but the full situation is a little more complex.

The tree of COHERENT directories in fact consists of any number of *file systems*, each of which exists on its own physical device. A *physical device* may be a partition on your hard disk, a floppy disk, or even a chunk of RAM.

The COHERENT system contains a suite of commands that let you create a new file system on a physical device, and graft (or *mount*) that new file system onto the COHERENT directory tree. The following few sub-sections will walk you through the steps of creating a new file system on a floppy disk and mounting it onto your existing COHERENT directory tree. These descriptions may be a bit too advanced for beginners; but most users will file them to be interesting and helpful.

fdformat: Format a Diskette

The first step in creating our new file system is to format a floppy disk. The command **fdformat** formats a diskette. When a diskette is formatted, COHERENT writes information on each track that makes it possible for the diskette to hold a file system.

fdformat uses the following syntax:

```
/etc/fdformat device
```

where *device* is the name of the device to be formatted. To format a high-density, 5.25-inch diskette, use the command:

```
/etc/fdformat /dev/fha0
```

To format a high-density, 3.5-inch diskette, type:

```
/etc/fdformat /dev/fva0
```

To format a low-density, 5.25-inch diskette, type:

```
/etc/fdformat /dev/f9a0
```

For this example, we'll assume that you have a high-density, 5.25-inch floppy disk. Insert into drive 0 (that is, drive A) of your computer, and type the command:

```
/etc/fdformat -v /dev/fha0
```

The **-v** option to **fdformat** tells it to verify that the disk is sound. This option means that the command will take longer to execute, but in the long run it's worth it as it will ensure that you do not waste time to trying to copy data onto a flawed disk.

When this command has finished executing, leave the floppy disk in drive 0.

See the Lexicon entry for **fdformat** for more information on this command and its options.

mkfs: Create a File System

The command **mkfs** creates a file system on a physical device. This command has the following syntax:

```
/etc/mkfs special proto
```

special names the physical device on which the file system is to be built. *proto* is either a number or a file name. If it is a number, **mkfs** builds a file system of that size in blocks.

For our example, type the command:

```
/etc/mkfs /dev/fha0 2400
```

This command will write a file system onto device **/dev/fha0**, which in this case represents the floppy disk in drive 0 that we just formatted. The number 2400 represents the number of blocks that fits onto such a disk. Please note that the above example is for a 5.25-inch, high-density floppy disk. For directions on how to create a file system on a floppy disk of different size or density, see the Lexicon article on **floppy disks**.

If *proto* is not a number, **mkfs** assumes that it is a prototype file. The command **badscan** scans a physical device for bad blocks and writes such a prototype file for you. Prototype files are beyond the scope of this example; but for information on them see the Lexicon entry for **badscan** or the Lexicon entry for **floppy disks**. The latter article summarizes all the ways in which floppy disks are used by the COHERENT system.

mount: Mount a File System

Now that you have formatted your floppy disk and built a file system on it, you can *mount* the newly created file system. *Mounting* grafts this device's file system onto the COHERENT system's directory tree. Thereafter, you can write files onto that device, read them, remove them, or do anything else that you wish with that device and its contents.

mount has the following syntax:

```
/etc/mount device directory
```

device names the physical device whose file system is to be mounted. *directory* names the *base directory* for that file system. The base directory is the directory by which the file system is accessed. For example, directory **/usr** is the base directory for the file system that holds all users' home directories. We'll describe base directories a little further in a few paragraphs.

For purposes of our example, type the following command:

```
/etc/mount /dev/fha0 /f0
```

This mounts the file system on the disk in drive 0 onto base directory **/f0**.

The base directory by convention is a directory in the root directory '/'. You do not have to do this, however. For example, if your user name was **henry** and you wished to mount the file system on the floppy disk in your home directory, you could type:

```
/etc/mount /dev/fha0 /usr/henry/backup
```

This will mount the file system on the floppy disk onto directory **/etc/henry** and name its base directory as **backup**. Note that if directory **backup** already existed in directory **/usr/henry**, its contents will be inaccessible until you unmount the file system on the floppy disk. Unmounting is discussed in the following sub-section.

For more information on mounting a file system, see (surprise!) the Lexicon article **mount**.

Using a Newly Mounted File System

Now that you have created and mounted a file system, you can use it like any other directory. To see how this works, type the following command:

```
cat >/f0/testfile
Here's some text we're writing onto the
newly mounted file system on a floppy disk.
<ctrl-D>
```

Here's you can use the **cat** command to write some text into file **testfile** which lives on the floppy disk you just mounted. To see that this text has been written there, type:

```
cat /f0/textfile
```

You should see the floppy-disk drive whirl briefly, and the following appear on your screen:

```
Here's some text we're writing onto the
newly mounted file system on a floppy disk.
```

You can now use this file system like any other, even though it lives on a floppy disk rather than your hard disk. As you can see, this is an easy way to extend the size of your COHERENT system's file system.

umount: Unmount a File System

Finally, when you have finished working with a file system, you must use the command **umount** to un-mount it. This command prunes the file system on a given physical device from the COHERENT system's directory tree. You will use this command frequently as you use floppy disks.

umount takes one argument: the name of the physical device being unmounted. In our example, the command

```
/etc/umount /dev/fha0
```

unmounts the file system on the high-density, 5.25-inch floppy disk insert into drive 0 (that is, drive A) on your computer.

Under unsophisticated operating systems like MS-DOS, you can insert or remove floppy disks without giving the matter a second though. The COHERENT system, however, uses a complex set of buffers to speed the reading and writing of information to the floppy disk; for this reason, if you simply yank a floppy disk out of its drive, all of the information in the COHERENT system's buffers will be lost. Worse, if you yank out a floppy disk and insert COHERENT-formatted floppy disk, the COHERENT system will write the data in its buffers onto that new floppy disk — and probably destroy its file system in the process. Unmounting a file system tells the COHERENT system to flush all information in its buffers and write it onto the disk.

To emphasize this point, please read the following carefully:

If you mount a floppy disk, you must use the **umount** *command to unmount it before you remove the disk from its drive. If you do not, data will be destroyed.*

This concludes the discussion of how to mount create a file system, mount it, and use it. See the Lexicon article **floppy disks** for further information on how to do this task.

The following two sub-sections discuss how to check a file system, to ensure its integrity.

fsck: Check a File System

The command **fsck** checks a file system, to ensure its integrity. For example:

```
fsck /dev/root
```

where **/dev/root** is a disk device, checks the file system located on device **/dev/root**.

If possible, you should **umount** the file system before you check it. You cannot **umount** the root file system. If you can't unmount it, be sure that no other users are on the system (i.e., that you are in single-user mode), then reboot the system immediately *without* performing a **sync**. If other users are creating or expanding files while the file systems are being checked, **fsck** will report false errors.

If **fsck** finds any discrepancies, it writes appropriate messages on the terminal. An absence of messages indicates that there are no problems with the file system. The appendix to this manual gives all of **fsck**'s error messages, and suggests how you should respond to each.

COHERENT's boot routines run **fsck** automatically, and will rerun it if necessary to fix problems with the file system. For more information on **fsck**, see its entry in the Lexicon.

Devices, Files, and Drivers

The next few sub-sections introduce the topic of special files and devices. You brushed this topic in the earlier section that described how to format and mount a file system on a floppy disk; the following few sections go into it more systematically. Beginners will probably find that much of this sub-section is mystifying, but experienced users and ambitious beginners probably will find much of value here.

To begin, the COHERENT system is designed to provide device-independent I/O. Devices and files are handled in a consistent way. Each I/O device is represented as a *special file* in directory **/dev**. For example, if your system has a line printer device named **lp**, you can list a file, named **prog** for example, on the printer by saying:

```
cat prog >/dev/lp
```

Another example is to copy the file **prog** with the **cp** command to your terminal:

```
cp prog /dev/tty
```

There are two types of special files represented in **/dev**, and when you list **/dev** with **lc** it will separate them.

The first type is a *block special* file. This type includes disks and magnetic tape. These devices are read and written in blocks of 512 bytes, and can be randomly accessed. (As a practical note, note that magnetic tape can be read in a random fashion only by positioning backwards and forwards one record at a time; disks can be read or written in a totally random fashion.)

The I/O to and from block devices is buffered to improve overall system performance. When a program writes a block of data, the data are held in a buffer to be written at a later time. If the same block is read twice in a row, the data for it is still available in memory and do not have to be fetched from the physical device.

A special program named **/etc/update** forces all buffered data to the physical device periodically by calling the command **sync**, to protect against losing data in the case of an accident, such as a power failure. If you must bring the system down, you must force the latest data to be written by typing the command **sync**.

Character-Special Files

The second kind of special file is called a *character-special* file. Included in this class are devices that are not block special: terminals, printers, and so on. Disks and tapes can also be treated as character special files. For every block special file for a disk, such as

```
/dev/at0c
```

there is usually a character-special file:

```
/dev/rat0c
```

Character-special files are sometimes called *raw* files, hence the prefix **r** in **rat0c**. A raw file has no buffering or other intermediate processing performed on its information. This difference is an efficient benefit to the **dump**, **restor**, and **fsck** commands, which do their own buffering.

tty Processing

One special set of devices has other processing — the **tty** or terminal files. A terminal-special file with this special processing is called a *cooked* device. The processing includes handling the **kill**, **erase**, **interrupt**, **quit**, **stop**, **start**, and **end-of-file** characters. Processing can be disabled with the command **stty** so the program deals with the raw device. However, using a raw **tty** device generally has negative effects on performance of the COHERENT system.

A Tour Through the File System

Our introduction to COHERENT's system of files and directories concludes with a tour of the COHERENT file system. Much of this material has been described earlier.

General File System Layout

The base of the file system is the **root** directory, whose name is simply:

```
/
```

Most of the files in the root are directories. To list the files in the **root** directory, type:

```
lc /
```

/bin

Most of the commonly used commands are programs contained in **/bin**, such as the command **lc** used in the above example. Foreign commands, such as MicroEMACS and **kermit**, are placed in directory **/usr/bin**.

The shell does not automatically look in **/bin** for commands, but consults the variable **PATH** to determine where commands are to be found. A typical value for **PATH** is:

```
/bin:/usr/bin:.
```

This tells the shell to look for commands in three places (in this order): **/bin**, **/usr/bin**, and finally **.**, the current directory. The shell does not consult **PATH** if the command contains one or more / characters, indicating a complete or partial path specification.

/dev

Devices in the COHERENT system are accessed through files in the directory **/dev**. If there is a line printer available on the system named **lp**, you can print characters from a file named **testdata** by typing the command:

```
cat testdata >/dev/lp
```

All devices on the system are represented in the **/dev** directory. Note that it is not recommended you access devices directly, but use the COHERENT system's utilities that *spool* files to them. This will prevent two users attempting to write material to a device simultaneously, and so garbling the output. For example, to access the line-printer device, use the spooler **lpr**. See the Lexicon's entries on **lpr** and **device drivers**.

/drv

A unique feature of the COHERENT system is the concept of loadable device drivers. This feature lets COHERENT system programmers write their own device drivers without modifying the rest of the system. Drivers can be unloaded, modified, and reloaded without halting and rebooting the system. Loadable drivers are kept in directory **/drv**. To load a driver, type:

```
/etc/drvld /drv/driver
```

where *driver* is the driver to load. See the Lexicon's entry on **drvld** for more information.

/etc

Several commands that you will use in your role as system administrator are kept in directory **/etc**. These are described in detail elsewhere in this guide. They include commands for system accounting, booting the system, mounting the system, create file systems, and control system time.

Also in **/etc** are several data files used in system administration. These include **/etc/passwd**, the file containing user names, ids, and passwords; news files; and file **/etc/ttys**, which describes the properties of each user terminal attached to the system.

/lib

The COHERENT system provides many useful functions for performing input and output (I/O) and mathematics, for use in your C programs. These and other libraries, along with the phases of the C compiler itself, are kept in directory **/lib**. This directory includes files containing standard system calls, standard I/O, and mathematical routines such as **sin**, **cos**, and **log**.

/usr

The directory **/usr** contains user directories, along with a few system directories.

/usr/adm contains additional information of interest to the system administrator.

/usr/bin contains commands that were not entirely created by Mark Williams Company.

/usr/games contains computer games. **/usr/games/lib/fortunes** holds a set of *bon mots*; the game **fortune** selects one at random and prints it on your screen. A call to this game can be placed in a user's **.profile**, so he will see a new fortune each time that he logs on. To add fortunes of your own, just edit the file **/usr/games/lib/fortunes**.

The directory **/usr/include** contains header files for C programs, such as **stdio.h**. Other header files define formats of files and other important data structures in the system.

/usr/lib contains the macro files **ms** and **man** used the **nroff** text processor; the unit conversion tables for the command **units**; and the file **/usr/lib/crontab** used to hold commands for **cron**. This directory also holds the C libraries.

/usr/man contains manual sections referenced by the commands **man** and **help** commands.

/usr/msgs stores messages displayed by the command **msgs**.

/usr/pub contains public files, such as telephone numbers and a copy of the ASCII table.

/usr/spool contains information for line-printer spooling, and mail that has not yet been delivered.

/u

In some systems, users' directories are placed on a separate device to save space. Because a separate device has a separate file system, the directory on that device is called **/u**.

Files: Conclusion

This concludes this tutorial's discussion of files and directories. The rest of this tutorial introduces COHERENT's suite of commands, and discusses topics of special interest to persons who are administering COHERENT systems.

Introduction to COHERENT Commands

This section introduces COHERENT's commands. The COHERENT system comes with more than 200 commands, which perform a variety of work, from formatting text, to editing files, to performing low-level administration of the system. The commands that manipulate files and directories were introduced in the previous section; there are, however, many other varieties of commands, many of which will be introduced here. To begin, we'll introduce the COHERENT system's master command, the *shell*.

The Shell

When you type commands into the COHERENT system, it appears that you are communicating directly with the computer. This is not exactly true, however. When you type into the COHERENT system, you are actually working with a special COHERENT program, the shell. This program reads, interprets, and executes every command that you type into the system. The shell can also interpret, expand, and otherwise flesh out what you type; this is done to help spare you unnecessary typing, and to permit you to assemble powerful commands with only a few keystrokes.

Please note, in passing, that the COHERENT system comes with two shells: the Korn shell **ksh** and the Bourne shell **sh**. These shells have somewhat different features. The descriptions in this section assume that you are using **sh**, which is COHERENT's default shell.

The shell is so powerful that mastering it is a major accomplishment; however, you can take advantage of much of what the shell offers by learning a few simple commands and procedures.

This section introduces some commands commonly used by COHERENT users. For more information on these or other commands see **help** and **man**. Also, consult the Lexicon.

Please note the following special punctuation characters:

```
* ? [ ] | ; { }
( ) $ = : ' ' " < > << >>
```

These characters have special meaning to the shell, and typing them can cause the shell to behave quite differently from what you may expect. Do not use these characters until you have read the following section, which discusses their use, or until they are presented in examples.

Redirecting Input and Output

Most COHERENT commands write their output to the *standard output* device, which is normally your terminal's screen. For example, **who** prints on your terminal the name of each user currently logged into your COHERENT system:

```
who
```

By using the special character >, you can redirect the output of **who** into a file. The

command

```
who >whofile
```

writes this information into **whofile**. The operator > tells COHERENT to *redirect* the standard output. Later, you can list the information on your terminal using **cat**:

```
cat whofile
```

Once the information is in a file, you can process it in other ways. For example

```
sort whofile
```

sorts the contents of **whofile** and prints the results on your screen. In this way, you can display the users' names on your terminal in alphabetical order.

You can also redirect the *standard input* to accept input from a file rather than from your terminal. To redirect the standard input, use the special character < before the name of the file that you want read as the standard input. For example, the command **mail** sends electronic mail to another user; normally, it "mails" what you type on the standard input, but you can use '<' to tell it to mail the contents of a file instead.

```
mail fred <whofile
```

mails the contents of **whofile** to user **fred**.

Pipes

The *pipe* is an important feature of the COHERENT system. Pipes allow you to hook several programs together by redirecting the output of one into the input of the next. A pipe is represented by the character '|' in the command line.

Most COHERENT programs are written to act as *filters*. A filter is a program that reads its input one line at a time or one character at a time, performs some transformation upon what it has read, and then writes the transformed data to the standard output device. You can easily perform complex transformations on data by hooking a number of simple filters together with pipes. Consider, for example, the command:

```
who | sort
```

Here, the command **who** generates a list of persons who are logged into the system. The output of **who** is then piped to the program **sort**, which sorts the list of users into alphabetical order and prints them on the standard error device.

The power and flexibility of the COHERENT operating system owes much to the pipe.

Superuser

A special user in the COHERENT system, called the *superuser*, has privileges greater than those of other users. The superuser can read all files (except encrypted files) and execute all programs. You must be logged in as the superuser during certain phases of your work as system administrator.

There are two ways to access the COHERENT system as the superuser. The first is to login under the user name **root**. When the system prompts

```
Coherent login:
```

reply:

```
root
```

This automatically makes you **superuser**. To remind you that you are superuser, the COHERENT system prompts you with **root:** instead of the usual **$**.

The second way to acquire the privileges of superuser is to issue the command

```
su
```

when you are logged in as a user other than **root**. You must have privileges to access **root** to do this, and you must know the password for **root**. When you type

```
<ctrl-D>
```

in this mode, COHERENT returns you to your previous identity.

To be the superuser for only one command, use the form of the command

```
su root command
```

command is the command to be executed as superuser. For example, to edit the message of the day file **/etc/motd** if you are not the superuser, type

```
su root me /etc/motd
```

When you finish using MicroEMACS, your original user id will be unchanged.

To limit access to privileged resources, the COHERENT system requires users to enter *passwords* before being granted that privilege. Users may be required to enter passwords before logging in.

If the **root** user has a password, you will be prompted for it. If you do not enter it correctly, the system will tell you

```
Sorry
```

and not allow you to become the **superuser**.

It is normal practice to protect access to superuser status by setting the password. If you are the only user of your COHERENT system, or if you deeply trust all other users, you do not have to do so. However, because the superuser can perform any sort of mayhem on your system, it is advisable to set the password, especially if outsiders can dial into your system via modem.

Manipulating Text Under COHERENT

The COHERENT system includes a number of commands and utilities with which you can process text. The phrase *process text* means to edit it and prepare it for printing.

MicroEMACS: Text Screen Editor

COHERENT includes a full-featured screen editor, called MicroEMACS. MicroEMACS allows you to divide the screen into sections, called *windows*, and display and edit a different file in each one. It has a full search-and-replace function, allows you to define keyboard macros, and has a large set of commands for killing and moving text.

Also, MicroEMACS has a full help function for C programming. Should you need information about any macro or library function that is included with COHERENT, all you need to do is move the text cursor over that word and press a special combination of keys; MicroEMACS will then open a window and display information about that macro or

function.

For a list of the MicroEMACS commands, see the Lexicon entry for **me**, the MicroEMACS command. A following section of this manual gives a full tutorial on MicroEMACS. In the meantime, however, you can begin to use MicroEMACS by learning a half-dozen or so commands.

To invoke MicroEMACS, type the command

```
me hello.c
```

at the COHERENT prompt. This invokes MicroEMACS to edit a file called **hello.c**. Now, type the following text, as it is shown here. If you make a mistake, simply backspace over it and type it correctly; the backspace key will wrap around lines:

```
main()
{
        printf("hello, world\n");
}
```

When you have finished, *save* the file by typing **<ctrl-X><ctrl-S>** (that is, hold down the control key and type 'X', then hold down the control key and type 'S'). MicroEMACS will tell you how many lines of text it just saved. Exit from the editor by typing **<ctrl-X><ctrl-C>**.

Now, re-invoke MicroEMACS by typing

```
me hello.c
```

The text of the file you just typed is now displayed on the screen. Try changing the word **hello** to **Hello**, as follows: First, type **<ctrl-N>**. That moves you to the *next* line. (The command **<ctrl-P>** would move you to the *previous* line, if there were one.) Now, type the command **<ctrl-F>**. As you can see, the cursor moved *forward* one space. Continue to type **<ctrl-F>** until the cursor is located over the letter 'h' in **hello**. If you overshoot the character, move the cursor *backwards* by typing **<ctrl-B>**.

When the cursor is correctly positioned, delete the 'h' by typing the *delete* command **<ctrl-D>**; then type a capital 'H' to take its place.

With these few commands, you can load files into memory, edit them, create new files, save them to disk, and exit. This just gives you a sample of what MicroEMACS can do, but it is enough so that you can begin to do real work.

Now, again *save* the file by typing **<ctrl-X><ctrl-S>**, and exit from MicroEMACS by typing **<ctrl-X><ctrl-C>**.

Just as a reminder, the following table gives the MicroEMACS commands presented above:

<ctrl-N>	Move cursor to the *next* line
<ctrl-P>	Move cursor to the *previous* line
<ctrl-F>	Move cursor *forward* one character
<ctrl-B>	Move cursor *backward* one character
<ctrl-D>	*Delete* a character
<ctrl-X><ctrl-S>	*Save* the edited file
<ctrl-X><ctrl-C>	Exit from MicroEMACS
<ctrl-Z>	Save a file and exit

Note that on some terminals, the arrow keys will not work. Note, too, that some remote

terminals may have trouble using **<ctrl-S>**, if they use XON/XOFF to control flow. In this case, use **<ctrl-Z>** instead.

For more information, see the tutorial for MicroEMACS included with in this manual.

pr, prps, lpr: Print Files

The command **lpr** prints files for you, making sure that your request does not conflict with other uses of the printer. To print a file, type the command

```
lpr file
```

substituting the name of the file to be printed for "file". Normally, the system prints a banner page before it prints a job; if you wish to suppress the banner page, use the **-B** option:

```
lpr -B file
```

If no file is given, the standard input is printed. Thus, **lpr** can be used in pipes; this allows you to print immediately matter that you type on your keyboard.

lpr will take your file and try to print it on any printer you have plugged into your computer's parallel port. If you do not have a printed plugged in, or if it is not turned on, **lpr** will hold onto your files until the printer becomes ready; it will wait days, if necessary, until the printer becomes available.

lpr is also intelligent enough to handle requests from several different users: if more than one user wants to print a file, **lpr** will print them one at a time. In this way, the COHERENT system lets several users share one printer.

lpr does nothing to the file other than print it. This means that no page headings are printed, nor does it break it the file up neatly into page-sized chunks. Another command, **pr**, does this for you. It paginates the standard input, giving a header with date, file name, page number, and line numbers. The paginated output appears on the standard output.

To print a paginated file on the line printer, type:

```
pr file | lpr -b banner
```

Note the use of the pipe '|', which passes the output of **pr** as input to **lpr**.

nroff, troff: Text Formatters

The commands **nroff** and **troff** format text for display or printing. They are, in fact, text-formatting languages: you type commands into your text file, and **nroff** or **troff** interprets the commands to format the text in the manner that you want.

nroff and **troff** differ in the style of formatting that they perform. **nroff** formats text into monospaced font, like that on an ordinary typewriter. Its output is suitable for display on the screen. **troff** formats text into proportionally spaced fonts, like those seen on this page. Its output is suitable for printing on a laser printer or other sophisticated typesetting device. The commands for **nroff** and **troff** closely resemble each other. The following descriptions will assume that you are using **nroff**, but they apply to **troff** as well.

nroff's programming language is quite complex and sophisticated. This manual includes a tutorial that introduces **nroff**'s language. You can, however, use **nroff** to perform simple formatting tasks by using the **ms** macro package. The following describes some of the more commonly used **nroff** commands.

To see how **nroff** works, type the following script:

```
cat >script.r
.ds CF "Print on Bottom of Each Page"
Here is some text.
Here is some more text.
.PP
The above command set a new paragraph.
Yet more text.
.SH
Here is a Section Heading
.PP
More text.
\fBThis is printed in bold face.\fR
This printed in Roman.
\fIThis is printed in italics or underlined.\fR
.PP
Here's some more text.
Here's yet more text.
And more text yet.
<ctrl-D>
```

Now, format and display the text with the following command:

```
nroff -ms script.r | more
```

You will see the text formatted for your screen. The string **Print on Bottom of Each Page** appears at the bottom of the display. The following describes the **nroff** commands with which this formatting was performed.

nroff's commands are introduced in either of two ways: by a period '.' in the *first* column of a line; or by a backslash '\' occurring anywhere in a line. The following reviews this script in detail.

.ds CF This defines the text to appear on the bottom of each page. If the text is more than one word long, it must be enclosed within quotation marks.

.PP Begin a new paragraph. **nroff** skips one line and indents the following line by five spaces (one-half inch).

.SH Print a section heading. **nroff** skips one line and prints in boldface the line of text that follows this command.

\fB Print the following text in **boldface**.

\fR Print the following text in Roman.

\fI Print the following text in *italics*.

With these few commands, you can perform simple formatting of your text.

To print the formatted text on an dot-matrix line printer, use the command **lpr**; to print it on a Hewlett-Packard LaserJet printer, use the command **hpr**. For example, to print **script.r** on a line printer, use the command:

```
nroff -ms script.r | lpr
```

To print **script.r** on a laser printer, use the command:

```
nroff -ms script.r | hpr -B
```

The **-B** option to **hpr** tells it to *not* print a banner page.

This discussion is sufficient to get you started, but it just scratches the surface of what you can do with **nroff** and **troff**. See their respective entries in the Lexicon for details of what these commands can do. See the tutorial for **nroff** that appears later in this manual for a thorough introduction to the formatting language used by these commands.

Miscellaneous Commands

COHERENT includes numerous commands that perform miscellaneous tasks. These include some of the most useful, and entertaining, commands in the COHERENT system.

who: Who Is On the System

To find who is logged into the system, use the COHERENT command **who**. This command lists who is logged into the COHERENT system, one name per line. You will see your own user name there as well.

If you sit down at a terminal that is not in use, but at which someone has already logged in, the following command tells you who is logged in:

```
who am i
```

COHERENT replies with the name of the user logged in at that terminal.

write: Electronic Dialogue

The command **write** lets you carry on a "conversation" with another user. The conversation continues until you or the other user type **<ctrl-D>** on his terminal.

For example, user **fred** can begin a conversation with user **anne** by typing:

```
write anne
```

On **anne**'s terminal, the message

```
Message from fred...
```

will appear. To establish the other half of the communication, **anne** should then say

```
write fred
```

and a similar notification appears on **fred**'s terminal.

At this point, both users simply type lines on their terminal and **write** sends the message to the other user. To avoid typing at the same time, each user should end a "speech" by typing a line that has the single letter

```
o
```

to signify "over", or "go ahead". When the other user sends you this, you know it is your turn to "talk", and vice versa.

When your communication is finished, you should type

```
oo
<ctrl-D>
```

Here, **oo** means "over and out", and the **<ctrl-D>** terminates the **write** command. Note that **o** and **oo** are polite conventions, and are not necessary to using **write**.

mail: Send an Electronic Letter

You can send electronic mail to another user on your COHERENT system by using the command **mail**. This command works whether or not that person is logged into the system at the time you type your message. The message is stored in an electronic "mailbox", and the user will notified that a message is waiting for him the next time he logs into your system.

Before you can use **mail** on your system, you must run the program **uuinstall**. This program will ask you some questions about how you have configured your COHERENT system, and will write files of information that **mail** and the communications protocol UUCP need to deliver your mail. For detailed directions on how to run **uuinstall**, see the section *Installing UUCP* in the UUCP tutorial that appears later in this manual.

Among other things, this program will ask you to name your "site" and your "domain". Without going into too much detail at this point, the site is *nom de plume* by which your machine is known to other COHERENT or UNIX systems. Site names generally are not computer-ese; **conan**, **terminator**, **lepanto**, **chelm**, and **smiles** are all examples of site names. If you don't intend to communicate with other systems, use your first name as the site name. The domain is the name by which a group of related machines are together known. If you and a number of other local COHERENT systems wish to be known together, you can establish a domain and register it with the network. Domain names, too, should be descriptive. If you don't intend to use a domain, set the domain name to **UUCP**.

To mail a message to user **anne**, just type:

```
mail anne
```

mail immediately prompts you for a title for your message:

```
Subject:
```

You can type the message's subject, which will be used to title the message, or you can just press **<Return>**.

Once you have titled your message, type the body of the message. You can conclude your message in any of three ways: you can type **<ctrl-D>**, type a period '.' at the beginning of a line, or a question mark '?' at the beginning of a line. The first two methods end the message immediately; the last method, however, invokes an editor, and lets you edit the message further before sending it on to the intended recipient. Environmental variable **EDITOR**, if defined, selects the editor to be used.

For example, to send your message to user **anne**, you might do the following. First, invoke **mail**:

```
mail anne
```

Next, give your message a title:

```
Subject:  I'll be working late
```

Finally, type the body of the message:

```
I'll be working late.  I hope to get home before Catherine
and George go to bed.  Please remind Ivan and Marian to do
their homework.  Marian should remember to practice her
violin.
<ctrl-D>
```

If you wish, you can first type your message into a file and then mail it. For example:

```
cat >hb.msg
All come to the birthday party at four
next to the pump room.
<ctrl-D>
```

To mail the message to user **jill**, type:

```
mail jill <hb.msg
```

You can send a mail message to several users at one time by listing each user's name on the command line. For example, the command

```
mail jill jack ted barb <hb.party
```

mails the contents of file **hb.party** to **jill**, **jack**, **ted**, and **barb**. To illustrate the use of the mail command, send yourself a mail message. Type the following; substitute your user name for "you" in the mail command:

```
mail you
Subject:  test the COHERENT mail system
This is a note to
myself to test
mail.
.
```

If someone has sent you mail, the COHERENT system will tell you:

```
You have mail.
```

when you log in.

To receive mail, type the **mail** command with no parameters:

```
mail
```

If you have no mail, COHERENT will tell you:

```
No mail.
```

If you do have mail, the system will print each message on your terminal, along with the user name of the sender, and the date and time that the message was mailed.

After each message, the **mail** program types a question mark **?** and waits for your reply. You can type any of the following commands in reply to the prompt:

d Delete the message.

<Return>
Proceed to the next message.

s *file* Save, or copy, the message into *file*.

q Quit — exit from **mail** and return to the shell.

You will know that you are finished with all of your messages when **mail** sends you a **?** without typing anything before it.

mail can also send messages to other COHERENT or UNIX systems via the UUCP utility. See the accompanying tutorial on UUCP to see how you can set up **mail** to do this.

msgs: Cumulative Message Board

The message of the day is deleted when a new message is inserted. If a user does not log in for several days, the message of the day may no longer be there. For items that you want everyone to see, such as hours of operation or new operating procedures, you should use **msgs** instead of **motd**.

msgs helps users get all important messages, even if they don't log in every day. The system remembers which users have seen each message. After a user logs in, invoking **msgs** will show the number, date, and author of each message written since the user last logged in. Therefore it is easy for the user to stay up to date with the system-wide messages.

To add a message to the file, simply mail the message to **msgs**. To title the message, write it as the first line in the message, after the "Subject:" prompt from **mail**.

The home directory for **msgs** will grow over time, as more and more messages accumulate. Also, if a new user is enrolled on your COHERENT system, he may have to wade through several hundred messages when he first logs in. Therefore, you should purge the home directory for **msgs** every now and again; you may wish to throw away the announcements of office parties three Christmases ago, and save important information on diskette.

msgs keeps track of what messages each user has read by recording the number of the last message read in the file **$HOME/.msgsrc**. When each user logs on, his version of **.msgsrc** is inspected to determine the last message seen. If messages were added after that, **msgs** prints the ones the user wants to see, and then updates **.msgsrc**.

grep: Find Patterns in Text Files

The command **grep** lets you find lines that contain a *pattern* within one or more files. Patterns are sometimes called *regular expressions*.

To illustrate **grep**, create file **doc1** by typing:

```
cat >doc1
a few lines
of text.
<ctrl-D>
```

Then the command

```
grep text doc1
```

prints the second line of file **doc1**:

```
of text.
```

The first parameter to **grep** is the *pattern* for which you are looking; the rest of the arguments are the names of files to be examined. **text** is the pattern and **doc1** is the file.

To find if a particular user is on the system, pipe **who** into **grep**:

```
who | grep you
```

(Substitute the user name in question for **you**.) Try it with your user name. The pattern is **you**, but no file name is specified. **grep** reads input from the standard input, which in this example is connected to the output of the **who** command.

You can specify several files to be searched; simply put the additional file names after the first:

```
grep pattern doc1 doc2
```

Or, you can search all files in the current directory for the pattern with

```
grep pattern *
```

The asterisk will be interpreted to mean all files, and grep will look for *pattern* in each.

The search pattern can be a *pattern*. Patterns are fully discussed in the tutorial for **ed**.

You can also locate lines that do *not* contain given patterns by using the **grep** option **-v**.

```
grep -v bugs prog1 prog2
```

This command finds and prints all lines in files **prog1** and **prog2** that do not contain the pattern **bugs**.

date: Print the Date

The COHERENT system keeps track of the time and date. To find the date and time, use the command:

```
date
```

COHERENT responds with the day of the week, the month day and year, and the time of day.

Internally, the COHERENT system records the date and time as the number of seconds since January 1, 1970, 00:00:00 Greenwich Mean Time (GMT). This means that files created in one time zone and referenced in another time zone will bear the correct time. The time and date printed out is converted from the internal form to the local time.

passwd: Change Your Password

You should change your password from time to time, to ensure that no unauthorized person can gain access to your files (or to the system as a whole).

It is easy to change passwords on the COHERENT system: just type the command **passwd**. **passwd** first asks you for your current password (if you have one), and then asks you to enter your new password twice. Entering the new password twice helps ensure that the system gets the password as you want it. If you do not type it the same way both times, COHERENT will say:

```
Password not changed.
```

You must then begin again with the command **passwd**.

Be sure the password is something that you can remember. It is recommended that the password be at least six characters long. Do not write it down, but memorize it. You can use a four-letter password, but if you do, you should mix upper-case and lower-case letters to make it more difficult for outsiders to guess.

stty: Change Terminal Behavior

Because a wide variety of terminals can be used with the COHERENT system, you must pass some information to the COHERENT system so it can handle your terminal correctly.

The command **stty** describes the information COHERENT currently has for you; you can then use **stty** with arguments to change how COHERENT handles your terminal.

For example, COHERENT normally echoes each character you type, as you type it. However, if your terminal also echoes what you type, you will see double characters. To prevent this, issue the command:

```
stty -echo
```

The program **login** uses this feature when you type your password, to help keep it secret from anyone who is kibbitzing at your desk.

To set the echo feature again, type:

```
stty echo
```

When you first log in, the system presumes that your terminal does not directly handle the **tab** character, so when COHERENT sends a **tab** to your terminal it simulates it with spaces. If your terminal does handle tabs, issue the command:

```
stty tabs
```

The COHERENT system will no longer substitute spaces for tabs. To go back to substitution,

```
stty -tabs
```

The **<erase>** character lets you delete the previously typed character. The **<kill>** character lets you delete the line that you have been typing but have not yet finished. By default, COHERENT sets these to, respectively, **<ctrl-H>** and **<ctrl-U>**. To change them to, respectively, **<ctrl-E>** and **<ctrl-K>**, use the **stty** command as follows:

```
stty erase ^E kill ^K
```

The carat '^' tells **stty** that you want to specify a control character.

To reset erase and kill to the default values at login, the command

```
stty ek
```

suffices. This is equivalent to:

```
stty erase ^H kill ^U
```

To see what your current terminal parameter settings are, type

```
stty
```

with no arguments.

Scheduling Commands For Regular Execution

The command **cron** is a valuable tool for using your COHERENT system. With it, you can schedule commands to be executed, even in your absence.

To specify a command to be executed at some later time, simply enter one line of information in the file **/usr/lib/crontab**. You must be logged in as **root** to modify this file.

For example, assume that you want to greet user **norm**, if he is logged into the system on Monday morning. You can do this by sending him a message at 8:13 on Monday. Use MicroEMACS to add the following lines to the file **/usr/lib/crontab:**

```
13 8 * * 1    msg norm%You are sure in early!
```

The numbers and * at the beginning specify the time:

```
13 8 * * 1
```

The **13** means "13 minutes past the hour". (**cron** numbers the minutes zero through 59.) The **8** means "8 AM". (**cron** numbers the hours of the day zero through 23, with zero indicating 12 AM.) The positions containing * normally specify the day and month. The two * characters mean "any day" and "any month". Finally, the **1** means "day 1 of the week," which is Monday. (**cron** numbers the days of the week zero through six, with zero indicating Sunday.) The breakdown of this command is shown as follows:

minute	13
hour	8
day of month	* — all days
month	* — all months
day of week	1 — Monday

Because each entry in **crontab** must be on one line, the symbol **%** represents the beginning of the input string. If the information is too long for one line, enter a backslash character before the **<Return>** at the end of the line. The backslash tells **cron** to ignore the **<return>**.

With this information in the file, **cron** executes the command

```
msg norm
Am Monday!
```

at 8:13 every Monday morning.

cron expects time to be in the 24-hour clock, so 1 PM is represented as **13** hours. If you need to print a literal percent sign '%', precede it with a backslash:

```
\%
```

The times for **cron** commands can be even more complex than the numbers and * shown above.

You can express a range for any of the five parts of a time by separating two numbers with a hyphen. For example, to send user **marianne** a humorous message on week days, use the command:

```
59 11 * * 1-5 /usr/games/fortune | msg marianne
```

To list a choice of times, separate single numbers or ranges with commas but no spaces. To send notification about a meeting on Monday, Wednesday, and Friday at 3 PM, use:

```
0 15 * * 1,3,5        echo Meeting at 3:30 ... | mail fred anne joe
```

The time specification

```
0 15 * * 1,3,5
```

represents the time 1500 (3 PM) on every Monday, Wednesday, and Friday.

mail and **msg** are just some examples of commands that can be used with **cron**; many others can be used. For example, **cron** is commonly used to execute UUCP commands late at night, when telephone rates are low. See the Lexicon article on **cron** for more information about this command. If you wish to schedule commands to be run but not on a regular basis, use command **at**. See its Lexicon article for further details.

Managing Processes

A *process* is a command that is undergoing execution. Because COHERENT is a multitasking operating system, numerous processes can be undergoing execution at the same time. The following commands let you monitor and, within limits, affect the operation of the processes your COHERENT system is executing.

ps: List Active Processes

Each process in the system is assigned a number called the *process id*, or *PID*. Each user logged into the system has one or more processes. Except in special circumstances, the first process that he has is the shell, or command-line interpreter. The commands he types are run by the shell.

The shell normally waits for a command to terminate before it begins to process the next command. However, if you use the '&' operator, the shell creates simultaneous processes: that is, while it executes one command it will let you type another. Thus, you can execute two or more commands simultaneously.

You can examine the processes associated with your login, or all processes in the system, with the command **ps**. Type:

```
ps
```

The result will resemble:

```
TTY     PID
console 3937 -sh
console 4010 ps
```

The first column

```
TTY   .
console
console
```

names the terminal you are running on, in this case the console. This identifier is taken from the file **/etc/ttys**, with the prefix **tty** removed from name. The **tty** identifier is also printed by the command **who**. The second column

```
 PID
3937
4010
```

lists the corresponding process identifier (PID). The third column names each command and gives its parameters, if any:

```
-sh
ps
```

-sh represents the shell process, and **ps** represents the **ps** command itself.

To see all the processes, type:

```
ps -a
```

The result will resemble:

```
TTY PID
3a: 41 -sh
39: 42 -sh
32: 47 - 3
31: 48 - 3
34: 193 -sh
36: 634 -sh
3e: 1738 -sh
20: 2568 -sh
3e: 2581 su
3c: 6317 -sh
3c: 6322 su
3f: 7333 - P
35: 7789 - P
3c: 8058
3d: 9053 - P
33: 9076 - P
30: 9814 -sh
30: 9829 ps -a
```

This display will, of course, differ quite a bit from system to system and from minute to minute.

For a full description of all options to **ps**, see its entry in the Lexicon.

kill: Signal Processes

Occasions will arise when the system administrator must log other users out of the system. For example, you may need to bring the system down quickly; or perhaps a user forgot to log out before leaving the terminal and did not see your broadcast message requesting that all users log out.

The command **kill**, when used by the superuser, terminates processes. To log out a user whose shell has process number 300, use the command:

```
kill -9 300
```

You must be logged in as **root** or use the command **su** to **kill** a process that belongs to another user. Each user can kill all processes that he owns, including his own shell process (which automatically logs him out).

kill has other uses as well — see the Lexicon's entry for **kill** for more information.

Programming Under COHERENT

The COHERENT system provides a number of languages in which you can write programs.

The shells included with COHERENT — **sh**, the Bourne shell, and **ksh**, the Korn shell — not only process commands, but are powerful programming languages in their own right. For details on how to program in these languages, see their respective entries in the Lexicon; and see the tutorial *Introducing sh, the Bourne Shell*, which follows in this manual.

COHERENT includes a full-featured assembler, with which you can assembly your assembly-language programs. Assembly language is sometimes required for operations that require you to work very closely with the operating system or hardware. For more information on the COHERENT assembler, see the Lexicon entry for **as**.

Most programming that cannot be executed efficiently by a shell language is done in C, the language in which the COHERENT system was written. The COHERENT system comes with a full-featured C compiler, with which you can compile the program you write in that language. If you are new to C, the tutorial *The C Language*, which follows in this manual, will introduce you to it. The following sub-sections briefly describe the tools available under COHERENT with which you can write, compile, and debug your C programs.

Basic Steps in COHERENT Programming

The steps that are necessary to generate a program are:

1. Create the program source file
2. Compile the source program, correcting any errors
3. Test and debug the program
4. Run the program

If you have compilation errors in step 2, or program errors in step 3 or 4, return to step 1.

Use **ed** or MicroEMACS to build and change the source program, the **cc** command to compile the source program and produce an object program, and **db** to help debug the program. Although the C compiler provides a macro facility, other languages do not. Therefore, if the source program uses macros, you can use **m4** to expand the macros.

This section covers each of these steps and provides some example programs.

Create the Program Source

The first step is to use MicroEMACS, **vi**, **ed**, or some other editor to create the program's source file. Details on the use of **ed** and MicroEMACS are covered in their respective tutorials, which follow in this manual. Each editor's commands are summarized in its Lexicon article.

For the first program, try a simple program that prints a short message on your terminal. For the sake of simplicity, we'll enter text using **cat** instead of invoking an editor. To build the program, type the following:

```
cat > small.c
main ()
{
        printf ("The COHERENT operating system\n");
}
<ctrl-D>
```

The first line invokes the concatenation program **cat** to enter the program's source code. The **<ctrl-D>** signals that you have finished entering text.

The program itself begins with the special word **main** which defines a function and must appear in every C program. The parentheses, here with nothing between them, enclose any arguments that are passed to the function. They are required even if there are no arguments. The body of the program appears between the braces { and }.

The function **printf** is part of the standard library of C programs. It prints formatted information on the terminal. In this case it will produce the string enclosed between quotation marks. The special character string

```
\n
```

means "newline". Two lines of output to the terminal can be produced by

```
"line 1\nline 2\n"
```

as an argument to **printf**. This appears in the output as:

```
line 1
line 2
```

For a fuller introduction to the C language, see the tutorial on the *The C Language*, which follows in this manual.

cc: Compile the Program

The command **cc** compiles C programs. It executes all the parts of the C compiler and the associated linker **ld**. The linker combines pieces of programs and includes necessary elements from the library, such as **printf**. The linker is occasionally called from the command line, but only for more complex problems than you are trying here. To compile our test program, type the command

```
cc small.c
```

If the compiler detects any errors, it prints a message on the terminal, along with the line number that contains the error. You can use this line number to find the error with your editor and fix it. You can now use the program by simply typing:

```
small
```

The tutorial on *The C Language* describes **cc** in greater detail; also see its entry in the Lexicon for a full summary of its many capabilities.

m4: Macro Processing

To extend the capabilities of all languages, the COHERENT system provides a macro processor, called **m4**.

Program source for all languages consists of character strings. Macro processors perform string replacement, whereby a string in the input file may be replaced by another string. **m4** provides parameter substitution, as well as testing values of currently available strings and conditional processing. **m4** is unique in that you can rearrange large sections of the input text by using the macros. For more information on **m4**, see the tutorial *Introduction to the m4 Macro Processor*, which follows in this manual.

make: Build Larger Programs

All the examples of programs thus far have been self-contained. As programs grow larger, it is usual to divide the source program into smaller files. This simplifies editing, speeds compilation, increases modularity, and lets several different programs share common functions.

Thus, in developing the larger program, you may have several source files in your directory, possibly an header file or two, and the object files that result from compilation. From these are built the executable file that runs when you type its name.

To change or fix the program, you must edit the source programs or header files in question with **ed**, recompile the required source, and relink all the modules. But, with a change that affects several modules, it can be tricky to remember exactly which modules need recompilation, and it can be time-consuming to recompile all modules.

COHERENT provides a command called **make**, which solves this problem. **make** examines the time a file was last modified, and the time of modification of files that it depends upon, and performs the necessary compilation or other processing. (COHERENT file system directories contain the time that each file was created or modified.)

The tutorial *The make Programming Discipline*, which follows in this manual, fully introduces this powerful and useful program.

db: Debug the Program

The first and most critical step to debugging programs is to not put bugs in them! The methods of structured analysis, design, and programming, or the method of stepwise refinement can substantially reduce the number of errors in a program.

One can also place **printf** statements at strategic points throughout the program to display logic flow and key data values. These display statements should be designed so that they can be turned off for normal operation without removing them from the program.

On occasion, however, you may find that it is necessary to debug at the machine level. If you must, COHERENT's **db** will make it possible to do so.

db provides tools that make the machine program instructions visible in the most natural notation. That is, instructions are displayed in a fashion that resembles assembly language, numbers can be displayed in hexadecimal, octal, or decimal as needed, and strings of characters displayed in familiar graphic form. **db** can also patch a program to be run again, as well as to control the execution of a program with breakpoints and one step at a time.

Briefly, to use **db** on a program like our sample **small** above, use the command:

```
db small
```

Now you can inspect and display instructions and data in the system, control execution, and even change the instructions in the program if you are bold enough.

To examine a data segment location in the program, simply type the address of the location. **db** knows about symbols in the program, so if you want to examine the location corresponding to **main**, type:

```
main
```

db types out the value in hexadecimal or octal (depending upon which is appropriate for your machine).

You can expand the display command to print many locations at one time, and choose the format of printout. To print five locations interpreted as instructions, type

```
main,5?i
```

where the format character **i** follows the question mark indicating format, and 5 is the count of locations to be printed. To exit **db**, type

```
:q
```

For a complete list of the format that **db** recognizes, and other details about **db**, see its entry in the Lexicon.

Administering the COHERENT System

The COHERENT system can be used by many people at the same time. One person must coordinate its use, like a key operator does for an office copier. This person is called the *system administrator*, and he sees to it that the COHERENT system runs smoothly every day. The administrator can also customize the COHERENT system to the needs of an individual installation.

Although you may be the only person to use your COHERENT system, many of the ideas discussed here are important for making your system work at its best. Please spend a few minutes reading this manual to familiarize yourself with the elementary concepts of COHERENT system maintenance.

Adding a New User

Each user allowed to use your COHERENT system must have a *user name* and a *user id*; the user may also have a *password*. The user name is usually the user's initials or a nickname. The *user id* is an integer number used to identify the user internally to the system. As system administrator, you will assign both of these for each user. This section tells you how.

To log in to the system, a user must have an entry in the *password* file **/etc/passwd**. The password file contains each user's name, id, and password if any. As system administrator, you will maintain this file.

Likewise, each group of users is assigned a *group name*, as well as a *group id*. Groups are not necessary to use the COHERENT system, but some installations prefer to set up groups by project or department.

It is simple to add a new user to the system. The command **newusr** takes care of all the details, and makes an entry in the password file. You must be logged in as **root**. For example, to create an entry for a user named Henry, log in as **root**, and then issue the command:

```
/etc/newusr henry "Henry Smith" /usr
```

This creates an entry in **etc/password** for **henry**, creates his home directory in the **/usr** file system, creates all appropriate files for him (such as his **.profile** and his mailbox), and sets all permissions correctly.

System Security

One of the most important tasks in running your COHERENT system is maintaining its security. Basically, security means two things: keeping outsiders from logging into your system, and keeping your system's users from doing untoward things. This section describes some steps you can take to ensure that your system is secure.

Passwords

Passwords provide the first level of COHERENT system security.

For systems with passwords, each user with a password must type his password as part of the login process. If he enters the password incorrectly, he cannot log in.

Your system's administrator can assign a password when she creates a user's log-in account, as described above. If you do not assign a password, anyone will be able to log in as that user.

In any system with passwords, it is especially important to assign a password to the **root**, or *superuser*. If the superuser does not require a password, any user can log in as **root** and automatically have access to the powerful tools that control the operation of the system.

Any user with a password can restrict access to his files. Once you assign him his password, he can change it with the command **passwd**. However, because of higher privileges, **root** can always access everyone's files.

The passwords are kept in file **/etc/passwd**, with the rest of the user login information. Passwords are encrypted, so reading **/etc/passwd** will not reveal passwords.

File Protection

The second level of COHERENT system security is *file protection*. A user can set each of three categories of protection for each of his files. A standard protection, or *access permission*, is given to each file when it is created.

The three categories of permissions are for the user himself, for other users in his group, and for all other users. To see the levels of protection of your files, type the command

```
ls -l
```

For more details on the meaning of each column in this printout, see the Lexicon entry for the change-mode command **chmod**.

Encryption

The command **crypt** provides a third level of system security. It lets a user encode and decode information in a file. The superuser has access to every file in the system; so to protect sensitive information even from his prying eyes, a user can disguise it with encryption. Sensitive system information, such as passwords, are also encrypted for security purposes; and the **mail** command lets users send encrypted mail to each other. For details about encryption, see the entry on **crypt** in the Lexicon.

Dumping and Saving Files

This section discusses how you can copy files to floppy disk. You should do this regularly, both to free up disk space and to back up valued files to protect them against catastrophe.

There are two general strategies for dumping files.

One strategy uses the commands **ustar** or **cpio** to create archives of files on a floppy disk. This strategy is fine for systems that are used by a handful of users, and that are not used for "real-world" jobs, such as running a business.

The other strategy uses the command **dump** to implement a system of regular dumps. This strategy is preferred for systems that daily amass data of importance for a real-world job, such as running a business or managing a research project.

You should always have a system of back-ups for your system. Which strategy you use depends on how you are using your system. The following sub-sections describe how to implement each strategy of back-ups.

Please note that the following descriptions assume that you are using a 5.25-inch, high-density floppy disks set in drive 0 (drive A).

Back-ups Using ustar

This sub-section describes how to back-up files using the COHERENT command **ustar**.

The first step is to prepare floppy disks to receive files. Insert a 5.25-inch floppy disk into drive 0, and then type the following command:

```
/etc/fdformat -v /dev/fha0
```

The command **fdformat** formats the diskette, verifying that no media defects exist. You must perform this task of formatting a floppy disk only before you use it the first time.

The next step is to create an archive of the files you wish to back up. Use the portable archive command **ustar** to collect a mass of files into an archive on the floppy disks. For example, to archive all files in directory **source**, use the following command:

```
ustar cvf /dev/rfha0 source
```

The options **cvf** tell **ustar** to create an archive, run in verbose mode, and write the archive onto the device or into the file named in the next argument. **/dev/rfha0** names the floppy device onto which you wish to write the archive. Finally, **source** is the directory whose files you wish to back up.

Note that you can name any number of files or directories on the **ustar** command line. If **ustar** requires more than one floppy disk to contain the archive, you will be prompted to insert another formatted, defect free floppy.

To perform a listing of the contents of the newly created archive, enter

```
ustar tvf /dev/rfha0
```

The options **tvf** tell **ustar** to list the contents of the archive, run in verbose mode, and read the archive from the device or file named in the next argument.

To extract several files from the archive, enter a command of the form

```
ustar xvf /dev/rfha0 source/myfile 'source/*.c'
```

The options **xvf** tell **ustar** to extract or unarchive the specified files, run in verbose mode, and read the archive from the device or file named in the next argument. Note that the second file argument contains a "wildcard" character and thus must be quoted to prevent expansion by the shell.

For more information on how to use **ustar**, see its entry in the Lexicon.

Back-ups Using dump

The following sub-sections describe how to perform back-ups using the COHERENT command **dump**. Here, dumps should be performed monthly, weekly, and daily.

Each month, you should dump all information in the system. You should prepare at least three sets of floppy disks for the monthly saves, giving you three months of full backup. You will use the diskettes in rotation, with the oldest always used next.

Once a week, you should dump information in the system that is new or has been changed since the beginning of the month. You will need five sets of diskettes, since some months have five weekends in them.

Finally, every day you should save information that has been changed or is new since the beginning of the week. For these dumps, you will need five sets of diskettes: one for each working day. You will need extras in case of weekend work.

Label each set of disks carefully as *monthly*, *weekly*, or *daily*. Label the daily diskettes Monday through Friday, the weekly diskettes Week 1 through Week 5, and the monthly diskettes Month 1 through Month 3. When you do the dump, write the date on the label.

The following gives a step-by-step description of how to use **dump** to back up files.

1. Log into the system as **root**. You must have superuser privileges to perform a dump.
2. If you have not yet done so, use the command **fdformat** to format a set of diskettes, as shown above. With high-density, 5.25-inch diskettes, a rule of thumb is to prepare one diskette for each megabyte of data to be dumped.
3. Tell other users to log off the system by typing:

   ```
   /etc/wall
   Please log off.
   Time for file dump.
   <ctrl-D>
   ```

   If you are the only user of your system, you can skip this and the following step.
4. Be sure that all users are logged off the system by typing the command:

   ```
   who
   ```

This command lists the names of all users that are still on the system.

If they have not logged off in a few minutes, send another message. Repeat the process until **who** shows no users except yourself. Note that a user may have left for the day without logging out, even though this is not a recommended practice. If backups are performed each day at the same time, users will develop the habit of logging off in time.

5. When all others have logged off, execute **/etc/shutdown** as described above.

6. If this is the last workday of the month, perform a *monthly* dump. Insert the first volume of the correct monthly dump floppy disk into the floppy drive, after adding today's date to the label, and type the command:

```
dump 0ufS /dev/fha0 2400 /dev/root
```

This will dump all files on the first hard disk partition /dev/root to the 2400-block diskette **/dev/fha0**.

If more floppies are needed, the computer will ask you to insert them. Be sure to label each with its volume number.

7. If this is the last work day of the week, but not the last workday of the month, perform a *weekly* dump. Prepare the correct weekly dump diskettes, add today's date to the label, insert the first diskette, and type the command:

```
dump 6ufS /dev/fha0 2400 /dev/root
```

This will dump files that have been changed or created this month to the diskette.

8. If this is neither the last workday of the month nor the last workday of the week, you will perform a *daily* dump. Prepare the daily dump diskette with today's day of the week, add today's date to the label, insert the first diskette into the drive, and type the command:

```
dump 9ufS /dev/fha0 2400 /dev/root
```

This will dump files that have been changed or created this week to the floppy disk.

9. Type **sync** to ensure that all buffers are flushed.

10. When you are finished dumping data, reboot the system by typing the command:

```
/etc/reboot
```

For more information on how to use **dump**, see its entry in the Lexicon.

Restoring Information

If you find that a file has been inadvertently destroyed, you can restore the information to disk from backup floppy disk.

To restore a file from a compressed **tar** archive, use the following commands. First, select the appropriate back-up disk, insert it into its drive, and mount it with the following command:

```
/etc/mount /dev/fha0 /f0
```

Next, use the commands **zcat** and **tar** to extract the file you want. For example, if your archive is called **backup.tar.Z** and the file wish to restore is called **myfile**, use the following command to extract it from its archive:

```
zcat /f0/backup.tar.Z | tar xf - myfile
```

The **zcat** command reads the compressed archive without requiring that you uncompress it. The **tar** command reads the standard input (as indicated by the hyphen '-') and extracts **myfile** from what it reads.

Once you have extracted your file or files, you can unmount the floppy disk in the usual way and put it away.

To restore information from back-ups created with **dump**, the process is a little more complicated. Here, you must use the command **dumpdir**, to find the file in your archive, and the command **restor** to extract it from your archive.

To begin, you must first determine the date and time that the file was last known to be good. From this date, determine on which set of disks the file was last correctly dumped. Find the set of floppy disks labeled with that date, and mount the first one in the set. For example, if you wish to restore the file **myfile**, use the command:

```
dumpdir f /dev/fha0 | grep myfile
```

lists files dumped on the high-density, 5.25-inch floppy disk and uses the command **grep** to select **myfile**. The use of **grep** spares you the problem of reading possibly hundreds of file names before you find the one you want. You may have to examine several disks before you find the one that holds your file.

Once you have found **myfile**, type the following command to restore the file:

```
restor xf /dev/fha0 myfile
```

The file will be restored into the current directory. It will *not* be named **myfile**; rather, its name will be the that of the number of the i-node of the original file. You can then rename the restored file to its original name or to a new name of your choice.

Please note that, as previously mentioned, certain precautions must be taken when performing a mass restore to the root partition. See the Lexicon article for **restor** for further information.

Dump and Restoring Using cpio

cpio is a public domain program written by Mark H. Colburn for the USENIX association, which is included with the COHERENT system. This program performs mass dumps and restores of files using a universally recognized file format. In general, **cpio** is easier to use than **dump** and **restor**, and its output can be portable among other COHERENT and UNIX systems.

If you are considering setting up a system of daily back-ups on your system, you should consider using **cpio** instead of **dump**. For information on how to construct **cpio** commands, see the Lexicon article for **cpio**.

System Accounting

The COHERENT system provides two types of computer time *accounting* to help you track the use of the system. Three commands control the accounting and provide reports at various levels of detail.

Note that system accounting adds overhead to your system, because your system has to do more work to record everything it does, and because the accounting files can quickly grow to unmanageable sizes. System accounting is useful for COHERENT systems that are being

used by multiple users who must account for (i.e., pay for) their use of the system, or in other circumstances where it is important to note each user's activity. For most systems that support a handful of users, system accounting simply isn't worth the bother.

If, however, you decide that you need system accounting, read on.

ac: Login Accounting

Whenever a user logs into the COHERENT system, it records the user's name, the terminal number, and the date and time of the login. It also records when he logs out.

You can use this information to compute the time each user, or all users, were logged into the system. The command **ac** prints the total of all login times recorded in the accounting file. An example of the result is

```
Total:  8357:00
```

You can ask for a summary of total login times for each day by typing:

```
ac -d
```

An example result would be:

```
Friday November 13:
       Total:  53:08
Saturday November 14:
       Total:  75:36
Sunday November 15:
       Total:  73:15
```

Finally, you can summarize the times for individual users with the command:

```
ac -p jack ted fred
```

This will show the total login times for these users:

```
fred          1100:42
jack          910:41
ted           641:58
Total:        2653:21
```

Also,

```
ac -pd
```

gives the time for each user, for each day that he logged in.

Login accounting is not automatically operational. The login information is collected only if the file **/usr/adm/wtmp** exists.

To start login accounting if it is not working, type the command

```
>/usr/adm/wtmp
```

while logged in as **root**. This creates the file **/usr/adm/wtmp** if it does not exist (and destroys existing information if it does) and thereby enables login accounting.

To turn off login accounting while it is running, you can type:

```
rm /usr/adm/wtmp
```

After you activate login accounting, you should purge **/usr/adm/wtmp** periodically as it grows continuously, and on an active system will eventually consume much disk space. To purge the current information but leave accounting turned on, type:

```
>/usr/adm/wtmp
```

sa: Processing Accounting

While login accounting tells you how much time a user spends logged into the system, it does not tell you the individual commands used. *Process accounting* does so. Under COHERENT, each execution of each command constitutes a separate process. (COHERENT's ability to maintain a list of processes and swap each in and out of memory until all are executed, is what gives COHERENT its multi-tasking capability.) Process accounting records system time, user time, and real time for each command executed by each user on the system. The command **sa** reports this information for you, using a format that you set.

sa has several options, to generate different reports. When used with no options, **sa** lists the number of times each call is made, the total CPU time, and the total real time used by the command, ordered by decreasing CPU time. This is a summary by command; the following gives an example:

```
            #CALL  CPU    REAL
sh          61     1      832
ld          5      1      7
ar          5      0      1
ranlib      3      0      1
p           16     0      11
dld         2      0      1
lc          19     0      1
cc          4      0      8
atrun       43     0      1
find        1      0      0
ed          1      0      2
cat         4      0      1
rm          3      0      0
j           1      0      0
spin        2      0      1
grep        2      0      0
msg         4      0      0
ps          1      0      0
pr          2      0      0
watch       4      0      0
who         2      0      0
stty        3      0      0
chown       1      0      0
sort        1      0      0
mv          2      0      0
pwd         1      0      0
nm          1      0      0
df          1      0      0
ls          1      0      0
echo        3      0      0
accton      1      0      0
```

The listing will depend on what commands are used in your system, and the characteristics of your hardware. To summarize by user, use the **-m** option:

```
sa -m
```

The option **-l** separates CPU time expended by users from that expended by the system. This command

```
sa -l
```

produces:

```
            #CALL  USER   SYSREAL
sh          61     0      1832
ld          5      0      07
ar          5      0      01
ranlib      3      0      01
p           16     0      011
dld         2      0      01
lc          19     0      01
cc          4      0      08
atrun       43     0      01
find        1      0      00
ed          1      0      02
cat         4      0      01
rm          3      0      00
j           1      0      00
spin        2      0      01
grep        2      0      00
msg         4      0      00
ps          1      0      00
pr          2      0      00
watch       4      0      00
who         2      0      00
stty        3      0      00
chown       1      0      00
sort        1      0      00
mv          2      0      00
pwd         1      0      00
nm          1      0      00
df          1      0      00
ls          1      0      00
echo        3      0      00
accton      1      0      00
```

To list the user name and the command name, use **sa** with the option **-u**. No times or counts are given. The command:

```
sa -u
```

produces output of the form:

```
tj          p
tj          lc
tj          find
tj          pr
bin         lc
tj          spin
tj          sh
bin         cc
bin         cat
bin         ld
bin         dld
farl        who
farl        sh
```

This report has been truncated and edited to save space. In practice, it is longer. The **-u** option overrides other options.

Process accounting is on only if you turn it on. To turn on process accounting, type the command:

```
/etc/accton /usr/adm/acct
```

while logged in as **root**. The file **/usr/adm/acct** holds the raw accounting information.

To turn off process accounting, use the same command with no file name:

```
/etc/accton
```

If accounting is not on when you type this command, you will get an error message. No information is gathered when accounting is turned off.

When process accounting is in use, the file **/usr/adm/usracct** grows with each user command issued. You should regularly condense or remove the information, to keep the file from devouring all free space on your disk. To condense the information, invoke **sa** with the **-s** option. You must turn off accounting while condensing information.

The information summarized by user will appear in **/usr/adm/usracct**, and information saved by command is placed in **/usr/adm/savacct**. These summarized files are used in future requests to **sa**. After condensing, you can turn accounting back on.

Additional options give flexibility to the report. See the entry for **sa** in the Lexicon for additional details on these options.

Conclusion

The following sections of this manual give tutorials to teach you how to use many of COHERENT's tools and commands. The Lexicon contains brief synopses of all commands, library routines, system calls, and macros available under the COHERENT system. It also includes many technical references and definitions, to help you with terminology throughout this manual.

Section 3:

Introduction to the awk Language

awk is a general-purpose pattern scanning language available with the COHERENT operating system. **awk** performs pattern matching, string manipulation, record processing, and report generation.

The syntax for **awk** is simple. It uses only one kind of statement, consisting of one or both of two elements: a *pattern* and an *action*. Patterns select the data to be processed, and actions specify the function to be performed on the selected data.

This tutorial explains how to write **awk** programs to process input. It will teach you how to use the **awk** interpreter and how to create an **awk** program. It describes the basic function of printing and the specification of input and output field and record separators. It explains the pattern scanning capabilities of **awk**. Finally, it describes the actions **awk** performs in addition to printing, such as assigning variables, defining arrays, and controlling the flow of data.

Using awk

awk reads input from the standard input (entered from your terminal or from a file you specify), processes each input line according to a specified **awk** program, and writes output to the standard output. This section explains the structure of an **awk** program and the syntax of **awk** command lines.

Program Structure

The basic element of an **awk** program is a statement in the form:

```
pattern {action}
```

A program may contain as many sets of *patterns* and *actions* as you need to accomplish your purposes.

awk checks each line of input with the *patterns* specified for a match, one pattern at a time. Each time the line matches a *pattern*, **awk** performs the corresponding *action*. After **awk** has compared the line with each *pattern* in the program, **awk** tests the next input line against the *patterns*.

An **awk** program may specify an *action* without a *pattern*. When **awk** processes an *action* which has no *pattern*, each input line matches. Therefore, **awk** performs the *action* on every line of the input.

An **awk** program may also specify a *pattern* without an *action*. In this case, when an input line matches the *pattern*, **awk** prints the line to the standard output.

One of the special *patterns* that **awk** recognizes is the word **FILENAME**. This *pattern* causes **awk** to print the name of the file that it is reading. Other special *patterns* are discussed below.

Records and Fields

awk divides its input into separate records, and subdivides each record into fields. Records are separated by a character called the input record separator (**RS**), and fields are separated by the input field separator (**FS**).

The default input record separator is the newline character, so **awk** normally regards each input line as a separate record. Because the default input field separator is either the space or the tab character, white space normally separates fields.

In addition to input record and field separators, **awk** provides output record and field separators (**ORS** and **OFS**), which it prints between output records and fields. The default output record separator is the newline character; **awk** normally prints each output record as a separate line. The default output field separator is the space character.

To process input with a record separator other than the newline character, use the special **BEGIN** pattern (fully described below) with an *action* that assigns the desired record separator to the variable **RS**. For example,

```
BEGIN {RS = ":"}
```

changes the record separator to a colon. You may specify any one character as the record separator. Specifying the null string (**RS=""**) makes two consecutive newlines the record separator. If you include more than one character within quotation marks, **awk** ignores all characters after the first one.

To change the output record separator, assign the desired character to the variable **ORS**. The output record separator may be a single character or a string. For example, the following program assigns the string *****record end***** to **ORS**:

```
BEGIN {ORS = "***record end***"}
```

The variable **NR** gives you the number of the current record. In the following program, **awk** prints this number at the beginning of each record to make editing easier:

```
{print NR, $0}
```

Here is a program that prints the total number of records in the input file.

```
END {print NR}
```

awk can also use the record number in relational expressions. To select a particular record for printing (for example, line 6), use the following program:

```
NR == 6 {print $0}
```

which tells **awk** to print the whole record when the number of the record is equivalent to 6.

Each record is subdivided into fields. Within the record, you may refer to each field separately by the name **$***n*, where *n* is the field number. For example, the fourth field is called **$4**. The entire current record is called **$0**.

Like records, fields have a default separator. For fields, the default separator is white space for both input and output fields (usually spaces or tabs; newlines can separate fields when **RS** is null).

You may change the field separator (variable **FS**) in two ways. The first way is to specify the change within the **awk** program, as follows:

```
BEGIN  {FS = ":"}
```

The sample statement changes the field separator to a colon. When you specify several characters within quotation marks, each character becomes a field separator, and all separators have equal precedence. For example, you can specify commas, colons, and periods to separate fields. In the following program, **awk** looks for any of these separators, and breaks the record into fields at each occurrence of each character:

```
BEGIN  {FS = ",:."}
```

The second method of changing the field separator is to use a command-line argument. The command line method enables you to declare the field separator at the time you invoke **awk**. To show how changing the input field separator affects the output, consider the following record from the file "now":

```
Now is the time for all good men
```

and the **awk** statement:

```
{print $1,$2}
```

When the input field separator is the default, the result of the **awk** program is:

```
Now is
```

When using the same statement but setting **FS = "i"**, **awk** prints the following:

```
Now s the t
```

As the input field separator, 'i' is not printed; however, in its place a blank separates the two output fields. The first field consists of uppercase 'N', lowercase 'o' and 'w', and a space. The second field consists of the 's', a space, the word "the", and the 't' of time.

When you use an input field separator other than the default, the printed output can look confusing, as in the example above. However, you can change the output field separator by assigning a character or string to the variable **OFS**.

To indicate where fields are divided when the output is printed, you can assign a character such as * to **OFS** as follows:

```
BEGIN {OFS = "*"}
      {FS = "i" ; print $1, $2}
```

This program prints the following:

```
Now *s the t
```

Notice that a semicolon (;) separates two statements on the same line.

The variable **NF** contains the number of fields in the current record. In the following program, **awk** prints the number of fields at the beginning of each output record, telling you the number of elements in the record:

```
{print  NF,$0}
```

awk can also use the variable **NF** in relational expressions. For example, to print all records with ten or more fields, you could use this program:

```
NF >= 10 {print $0}
```

Command Line Arguments

As with any COHERENT program or command, you invoke **awk** by typing the lowercase letters **awk**. To process files with **awk**, you must include some additional elements on the command line, called *arguments*.

The complete form for the **awk** command line is:

awk [-**y**] [-**F***c*] [-*f progfile*] [*prog*] [*file1*] [*file2*] ...

Each argument is described below.

The -**y** option enables you to name *patterns* in lowercase characters, which **awk** then matches to both uppercase and lowercase characters in the input file. This option is similar to its counterpart in the regular expression pattern-matching utility, **egrep**.

The following programs show how the -**y** option works on the file named **the**, which contains the following two lines:

```
The time is right.
Now is the time.
```

Command	*Output*
`awk -y '/the/' the`	`The time is right.` `Now is the time.`
`awk '/the/' the`	`Now is the time.`

The option -**F***c* is the command-line version of

```
FS = "c"
```

which is an assignment like the one described earlier. This option changes the input field separator from the default (white space) to the character *c*. You may include any characters you want **awk** to use as field separators after the -**F** flag.

The -**f** *progfile* option enables you to use a file *progfile* containing **awk** commands as an **awk** program. The option flag (-**f**) must precede the name of the file to be used as a program.

If you do not use the -**f** *progfile* option, you must use the *prog* option. This option specifies the **awk** program on the command line. When writing a command-line **awk** program, use a apostrophe before the first statement (*pattern*, *action*, or both); then enter the subsequent lines of the program. After the last statement of the program, type another apostrophe mark followed by the file or files to be processed. Note that COHERENT prompts you to enter more information by displaying the '>' at the beginning of each line until you enter the closing apostrophe and newline character.

The following program is an **awk** command-line program. It prints a heading before **awk** reads the input file **test**, and then prints the entire file with each line preceded by its line number.

```
$ awk 'BEGIN {print "sample output file"}
>            {print NR, $0}' test
```

The *file1 file2* ... option enables you to process existing files. When you want to process more than one file, separate the file names with white space. If you do not specify a file name in the command line, **awk** takes input from the standard input.

awk -F: '{print "mv" $1, $1.".r"}' addrept

after deleting &c cohpage# > addnext

The following program prints the files **test1** and **test2**. Each line is preceded by its record number.

```
$ awk '{print NR, $0}' test1 test2
```

Printing with awk

Printing is an **awk** *action*. In fact, it is the action most often used, because it is the simplest to use. The following short **awk** program prints its entire input:

```
{print}
```

When you specify **awk** actions, you may include several actions within one set of braces; however, each action must be separated from the others by semicolons (;) or newlines.

Printing Individual Fields

With **awk**, you can print output fields in a different order from the input fields.

You can print fields in any order you desire. For example, you can print the second and third fields in reverse order:

```
{print $3,$2}
```

When this program processes the input file **now** containing the sample record used above, the printed result is:

```
the is
```

Because the field names are separated by a comma, **awk** inserts an output field separator between the fields when printing them.

If you do not separate field names by commas in the print statement, **awk** concatenates the fields when printing them. For example, the following program prints the second and third fields:

```
{print $2 $3}
```

The result is:

```
isthe
```

Changing the Output Field and Record Separators

You may change the output field separator by assigning your desired separator to the variable **OFS**. To use the same field separator with the entire input, make the assignment before the first print statement. For example, to make the colon your output field separator, use a statement like this:

```
{OFS=":"; print $2,$3,$4}
```

You will receive this output:

```
is:the:time
```

To change the separator for the first line only, use the statement:

```
NR ==1{  OFS=":";print $2,$3,$4}
```

To change the output record separator from the default newline, assign required separator to the variable **ORS** in the same manner.

Printing Predefined Variables

As discussed earlier, you can print either or both of the **NF** (number of fields) or **NR** (number of records) predefined variables. To print a predefined variable, simply name it in the print statement. For example, to include the **NF** variable before the other output in the previous example, edit the program to read as follows:

```
{OFS = ":"; print NF,$2,$3,$4}
```

The output resulting from this statement is:

```
4:is:the:time
```

You can specify the **NR** variable in the same way. When you add the name of the variable to the desired place in the list of fields to be printed, **awk** prints the record number in that place in the output.

Redirecting Output

In addition to printing to the standard output, you also may redirect output to a file or files of your choosing. This ability to direct output to any file enables you to extract information from a given file and construct new documents.

Suppose you have a file named **accounts** with accounting information stored in it. The first column of the file contains payroll information, the second column shows income for the year, and the third column reports accounts payable information. You are to make an income report for the year containing text and tables.

To extract the income information from the **accounts** file and put it into a separate file named **income**, you can use the following **awk** program:

```
{print $2 > "income"}
```

With this program, **awk** creates the file **income** if it does not already exist, and enters the second column of the **accounts** file as the contents of the new file. If a file named **income** already exists, **awk** replaces the current contents of the file with the second column of the **accounts** file.

If you need the first two columns for two separate reports, you can redirect both columns to separate files using one statement.

```
{print $2 > "income"; print $1 > "payroll"}
```

You can specify a maximum of ten files for output.

If text for your report is already contained in the file **report**, you can append the second column of the **accounts** file to the end of your report using this **awk** program:

```
{print $2 >> "report"}
```

Appending enables you to complete your report without retyping a column of numbers that exists in another file.

Formatting Output

When you use **awk** to process a column of text or numbers as in the example above, you may want to specify a consistent format for the output. The statement for formatting a column of numbers follows this *pattern*:

```
{printf "format", expression}
```

where *format* is prescribed by the format control characters and separators defined below. *expression* specifies the fields for **awk** to print.

The following table shows the names and meanings of the most frequently used **awk** format control characters. To be recognized as format control characters by **awk**, these characters must be preceded by the percent sign **%** and a number in the form of *n* or *n.m*.

Format-Control Characters	*Meaning*
%nd	Decimal number
%n.mf	Floating-point number
%n.ms	String

When you call the **printf** function through **awk** to format the output, you must specify the output separators you want to use.

Output-Separator Character	*Meaning*
\n	Newline
\t	Tab
\f	Form feed
\r	Carriage return
\"	Quotation mark

For example, if you wish to print a column of numbers with up to nine places to the left of the decimal and two to the right (for a total of 12 places, including the decimal), and you want a new entry for each line, use a format like this:

```
{printf "%12.2f\n", $2}
```

Piping Output

You can pipe the output of your **awk** program to another process. The pipe connects the standard output of **awk** to the standard input of another process, program, or utility.

For example, you can pipe output to the **mail** utility with the following program, which mails the output to **name**:

```
{print | "mail name"}
```

The pipe operator is the vertical bar character between the print and mail commands in this statement.

awk Pattern Scanning

The previous section described printing in terms of fields. Fields are generally the best way to select single elements from columnar input files. In addition to names of fields, **awk** can scan records for the following:

- Two special *patterns*: **BEGIN** and **END**
- Regular expressions
- Arithmetic relational expressions
- Boolean combinations of expressions
- Pattern ranges

Special Patterns: BEGIN and END

BEGIN is a special *pattern* that matches the beginning of the input, before **awk** processes any of the input. As mentioned above, **BEGIN** is the best place to set the field and record separators if you want the same separators for the entire input. **BEGIN** is also a good place to perform the *action* of assigning values to variables when the values are known.

Actions that require **awk** to compare input with the variable **NR** may not produce the results you expect from a **BEGIN** *pattern*, because all **BEGIN** processing is finished before **NR**=*1*. Also, **awk** does not permit field references in **BEGIN** or **END** statements.

END is a special *pattern* which matches the end of **awk** input. The **END** *pattern* enables you to request an *action* to occur when all processing is finished. A common use of **END** is printing the value of variables. For example:

```
END {print NR}
```

tells **awk** to print the value of **NR** after processing is finished, giving the total number of records processed. When you reach the **END** *pattern*, you may not return for further processing.

You may make **awk** into a calculator by using **END** with no *action*. At the end of the input, you may enter any arithmetic equation or **awk** function and have the result automatically printed on the standard output. When you are finished using **awk** as a calculator, type <ctrl-D>.

Patterns

You can enclose strings of characters in slashes '/' for **awk** to match, as **ed** (the COHERENT text editor) and **egrep** (the COHERENT text pattern matching command) do. For example, take this pattern:

```
/ted/
```

When a statement contains this expression, **awk** prints every record with the string **ted**, whether **ted** occurs as a word or as part of a word. For example:

```
interested
busted
tedious
```

In addition to specific strings, you can scan for classes and types of characters. To do so, enclose the characters within brackets, and place the bracketed characters between the slashes. For example, to specify a range of lowercase letters, enclose the range of letters

within brackets:

```
/[a-z]/
```

You can specify ranges of uppercase letters or numerals the same way.

In addition, you can use the following special characters for further flexibility:

[ ]	Class of characters
( )	Grouping subexpressions
|	Alternatives among expressions
+	One or more occurrences of the expression
?	Zero or more occurrences of the expression
*	Zero, one, or more occurrences of the expression
.	Any non-newline character

When adding a special character to a pattern, enclose the special character as well as the rest of the pattern within slashes.

To search for a string that contains one of the special characters, you must precede the character with a backslash. For example, if you are looking for the string "today?", use the following *pattern:*

```
/today\?/
```

When you need to find an expression in a particular field, not just anywhere in the record, you can use one of these operators:

~	Contains the data in question
!~	Does not contain the data in question

For example, if you need to find the characters **jam** in the fourth field of the input, you can use the following statement:

```
$4~/[Jj]am/
```

This statement prints all lines where the fourth field contains **Jam** or **jam**. The statement also prints lines where the fourth field contains words like **James**, **jammed**, and **pajamas**. To prevent the **awk** program from selecting lines with characters other than separators on either side of the required expression, use the following special characters:

^	Beginning of the record or field
$	End of the record or field

With these characters, you can be still more specific about which field or record you want printed. For example, to allow **James** to be printed, but not **pajamas**, use the following statement:

```
$4~/^[Jj]am/
```

To allow only **Jam** or **jam**, use this statement:

```
$4~/^[Jj]am$/
```

Arithmetic Relational Expressions

An **awk** *pattern* may consist of relational expressions using the following operators:

<	Less than
<=	Less than or equal to
=	Equivalent
!=	Not equal
>=	Greater than or equal to
>	Greater than

With these operators, you can select fields according to their relation to one another. For example, if you want to print the first field only when it does not equal the second field, use this statement:

```
$1 != $2 {print $1}
```

You also can establish relationships among records. If you want to print no more than the first ten records, use the following statement:

```
NR <= 10
```

Because this example specifies no action, the statement prints all the records whose record number is ten or less.

Relational tests default to string comparison if either operand is nonnumeric. Thus, if one operand is numeric and the other is a string, **awk** makes a string comparison. The following example shows how **awk** compares one field to part of the alphabet:

```
$1 <= "C"
```

This statement selects all lines beginning with an ASCII value less than or equal to that of the letter 'C' (octal 103).

When you compare fields that have numeric values to one another, **awk** performs a numeric comparison. Consider the comparison in this example:

```
$2 < $1 + 100 {print $2}
```

This statement causes field 2 to be printed only when the value of field 2 does not exceed the value of field 1 by 100. If field 2 is alphabetic, it always matches in this comparison because strings evaluate to 0 in numeric comparisons.

Boolean Combinations of Expressions

awk tests logical combinations of expresssions in its pattern-scanning process. Use the following operators for combining expressions:

||	Boolean OR
&&	Boolean AND
!	Boolean NOT

The following example tests for records that begin field 1 with a character that is less than **u**, greater than or equal to **t**, and begin field 1 with a string other than **the**.

```
$1 < "u" && $1 >= "t" && $1 != "the"
```

The effect of this *pattern* is to select records that have a **t** as the first character in field 1 but

do not begin field 1 with the letters **the**.

Pattern Ranges

awk may cause an *action* to be performed on all records between two specified *patterns*. For example, to print all records between the *patterns* **April 10** and **April 19** inclusive, enclose the strings in slashes and separate them with a comma; then indicate the **print** action, as follows:

```
/April 10/,/April 19/ {print}
```

You also may specify a range of record numbers using a statement such as this:

```
NR == 5, NR == 17 {print}
```

This statement specifies that records 5 through 17 of the input are to be printed.

Specifying awk Actions

This section describes **awk** *actions* other than printing *actions*. In addition to printing, **awk** is capable of:

- Performing functions
- Assigning variables
- Using fields as variables
- Concatenating strings
- Defining arrays
- Using control statements

Functions

awk includes functions that enable you to perform specific calculations with input information. You may assign these functions to any variable and use them in patterns. The following list shows the functions and their definitions; an argument can be any expression.

length Return the length of the current record.

length(*argument*)
Return the length of *argument*.

sqrt(*argument*)
Return the square root of *argument*.

exp(*argument*)
Return e to the power of *argument*.

log(*argument*)
Return the natural logarithm of *argument*.

int(*argument*)
Return the integer part of *argument*.

abs(*argument*)
Return the absolute value of *argument*.

substr(*str,beg,len*)
Return the substring of str that is *len* characters long beginning at position *beg*. When **substr** occurs in a statement, **awk** scans *str* for the position *beg* within the

string. When **awk** finds *beg*, it prints a substring *len* characters long starting at *beg*. If *len* is not included in the argument, the substring includes everything from *beg* to the end of the record.

index(*s1*,*s2*)
Return the position of *s2* within *s1*, or zero if *s2* does not occur in *s1*.

sprintf(*f*,*e1*,*e2*)
Return strings *e1* and *e2* in the **printf** format *f*

split(*str*,*array*,*fs*)
Divide *str* into fields associated with *array* (an array is a collection of fields listed under a single name) that are separated by *fs* or the default field separator.

The **sprintf** function lets you format expressions *e1* and *e2* according to format specification *f*. The following example demonstrates the operation of the **sprintf** function.

```
> awk 'x = sprintf("%7.2s",$1)
> {print $1}
> END {print x}'
```

When you run this sample program, **awk** accepts input data from the keyboard of the terminal. The first line of the program begins the **awk** program and sets variable **x** so that it contains five blank spaces and the first two characters of the first input field. The second line causes **awk** to print the first field as it was received. The third line ends the program by printing **x**, the formatted version of the first input field.

If you enter the word **chicago** as the first input field for this program, **awk** prints:

```
chicago
     ch
```

The **split** function divides fields into subfields, breaking *str* into elements of *array* separated by *fs*, or white space when *fs* is not specified. In the following example, **awk** splits the first field of the record into subfields. If the record has a single colon in the first field, **awk** splits the field into two subfields. These subfields become the first and second fields of the array named **time**:

```
{split ($1,time,":")}
```

At this point, you may manipulate the information stored in the array **time** or simply print the subfields.

Assignment of Variables

In addition to the intrinsic variables, such as **NR** (which contains the number of the current input record) and **FILENAME** (which contains the name of the current file), you may assign other variables as described below.

Variables in **awk** may be string or numeric variables, depending on the context. By default, variables are set to the null string (numeric value zero) on start-up of the **awk** program. To set the variable **x** to the numeric value one, you can use the following assignment statement:

```
x = 1
```

To set x to the string **ted**, use the following statement:

```
x = "ted"
```

When the context demands it, **awk** converts strings to numbers or numbers to strings. For example, the statement

```
x = "3"
```

assigns to **x** the string **3**. When an expression contains an arithmetic operator such as the '-', **awk** interprets the expression as numeric. (Alphabetic strings evaluate to zero.) Therefore,

```
x = "3" - "1"
```

assigns the value two to variable **x**.

When the operator is included within the quotation marks, **awk** treats the operator as a character in the string. In the following example

```
x = "3 -1"
```

assigns the string

```
"3 - 1"
```

to variable **x**.

You also can perform numeric calculations on fields. For example, you can calculate the sum of the fourth field in the following manner:

```
{sum += $4}
END {print sum}
```

The following table includes all the available operators for **awk**:

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
++	Increment
--	Decrement
+=	Add and assign value
-=	Subtract and assign value
*=	Multiply and assign value
/=	Divide and assign value
%=	Divide modulo and assign value

You may use any of these operators in **awk** expressions.

Field Variables

In **awk**, fields may receive assignments, be used in arithmetic, and be manipulated in string operations. The following **awk** statement shows some of the available uses of fields as variables.

```
{print $i, $(i+1), $(i+n)}
```

awk permits you to use numeric expressions to refer to fields. Here, print fields i, i+1, and i+n.

String Concatenation

As mentioned earlier, you can concatenate strings by omitting comma separators from printing actions. For example, the following print statement concatenates the first two fields by inserting a new connecting string:

```
{print $1 " telephones "  $2}
```

If $1 contains "Tom" and $2 contains "John", this statement prints:

```
Tom telephones John
```

Arrays

Under **awk**, an array is a collection of values that is labeled with the name of the array. Each element has at least one named **index**. The array is implicitly declared because **awk** creates the array when you name it. Also, you can name the individual indices with any legal string or numeric value.

Because the indices for any array may have any value, the ordering of array elements is arbitrary. However, when you use numeric index names exclusively, **awk** follows an ascending numeric sequence.

You should specify the array element using an identifier followed by the array index, an arbitrary expression enclosed in brackets ([]). For example, consider an array called **surname**. This example uses array indices named **tom**, **van**, and **gordon**. The following action assigns a value to each of these indices:

```
BEGIN  {surname ["tom"] = "jones"
surname ["van"] = "johnson"
surname ["gordon"] = "smith"}
```

You can print the contents of the array by naming the array in a **print** statement. **awk** also enables you to print the name of the index by associating another variable with the index, using a special form of the **for** statement. This form of **for** is:

```
for (index in array)
```

To retrieve the index names of the array **surname**, you may use the following statement:

```
END  {for (person in surname)
print person, surname[person]}
```

This statement yields the following output:

```
tom  jones
van  johnson
gordon  smith
```

In addition to being a generic term for the indices in the array **surname**, **awk** creates an array of names called **person**, to which you can make further associations as needed.

To store the number of occurrences of a pattern, you may use the associative array capabilities of **awk**. For example, if you want to determine the number of occurrences of **mark** and **test**, and print the number next to its respective word, you can use the following program:

```
/[Mm]ark/   {n["mark"]++}
/[Tt]est/   {n["test"]++}
END         {for (word in n)
            print word, n[word]}
```

With each occurrence of **Mark** or **mark**, **awk** increments the variable **n[mark]**. (**awk** automatically initializes **n[mark]** and **n[test]** to zero at the start of execution.) After **awk** processes the last line of the input, the program prints each word and the number of occurrences of that word as stored in **n[word]**.

Control Statements

awk has seven defined control statements. The following section explains the statements and gives examples of their use.

if (condition) else

If the *condition* within the parentheses is true, the statement following the **if** is executed. If there is a clear alternative, the **else** precedes the action to be performed when the condition is false. The **else** is optional. If **awk** does not perform the *action* of the **if** statement and there is no **else** statement, **awk** continues with the next statement. For example:

```
{
if (NR % 2 == 1)
        print "odd-numbered record"
else
        print "even-numbered record"
}
```

while (condition)

While *condition* remains true, the statement following **while** is executed. For example:

```
{
        i = 1
        while (i <= NF){
                print $i
                i++
        }
}
```

for

The **for** statement lets you execute actions a specified number of times. This statement may contain an initialization portion, a Boolean test, and an incremental counter. The initialization portion sets the initial value of the count variable, which **awk** changes each time it performs the action. The Boolean test defines the conditions under which **awk** should continue the action. The incremental counter specifies how **awk** is to alter the count variable each time it performs the action. For example:

```
{
for (i = 1; i <= NF; i++)
        print $i
}
```

break

The **break** statement immediately interrupts a **while** or **for** execution. For example:

```
{
for (i in numbers){
        if (numbers [i] == "stop")
                break
                print i, numbers [i]
        }
}
```

continue

The **continue** statement immediately begins the next iteration of the **while** or **for** statement. For example:

```
$1~ /Smith/  {
        for (i = 2; i <= NF; i++){
                if ($i < 100)
                        continue
                sum += $1
        }
}
```

next

The **next** statement causes processing to skip to the next record for comparison with all the *patterns*, beginning with the first, and in order. For example:

```
NR % 2 == 1{
        print "odd-numbered record"
        next
}
{        print "even-numbered record"
}
```

exit

The **exit** statement forces the **awk** program to skip any remaining input and to execute the *actions* at the **END** patterns. For example:

```
sum >= 1000  {exit}
       {sum += $4}
END          {print NR, sum}
```

For More Information

The Lexicon's article on **awk** gives a quick reference of its features and options.

Section 4:

bc Desk Calculator Language

This tutorial introduces **bc**, the calculator language for COHERENT. If you have not used **bc** before, this tutorial will introduce you to its features and functions. If you are familiar with **bc**, you can use it as a reference.

bc is a language that can calculate to high precision. It automatically adjusts the number of digits in a number to represent it correctly. It is like having a powerful calculator at your fingertips.

Entry and Exit

The **bc** calculator for COHERENT is easy to use. Whenever you wish to invoke **bc**, all you do is type its name (**bc**), followed by a stroke of the carriage return key. When you are finished using the calculator and wish to exit, just type the word 'quit' or **<ctrl-D>**. **bc** exits and return control to COHERENT.

Example of Simple Use

bc performs calculations on formulas that you type into it. The formulas are laid out as you would naturally write them. For example, to invoke **bc**, have it add 2+2, and then exit, type:

```
bc
2 + 2
```

bc replies:

```
4
```

Then, leave **bc** by typing:

```
quit
```

bc is an arbitrary precision calculator: the number of digits carried by **bc** depends upon the requirements of the calculation, and is automatically expanded by **bc**. Thus, **bc** will never overflow. The number of digits it carries is limited only by the amount of available computer memory. For example, try this calculation:

```
2^500
```

The carat '^' character signifies a superscript; thus, we are asking **bc** to raise 2 to the 500th power. After a moment, **bc** will reply:

```
327339060789614187001318969682759915221664\
204604306478948329136809613379640467455488\
327009232590415715088668412756007100921725\
654588539305332852758937 6
```

You have probably already noticed one nice thing about this calculator: you don't have to include a print statement as part of your command, because **bc** automatically prints the results onto your terminal screen. When **bc** sees any expression, like "2+2" or "3777", it

prints the result.

bc provides the common arithmetic operators for add, subtract, multiply, and divide, as illustrated by the following commands:

```
7 + 5
7 - 5
7 * 5
7 / 5
```

bc also provides the remainder operator '%'. To get a sense of how it works, type:

```
7 % 5
5 % 7
```

Here, **bc** prints the *remainder* of the first number divided by the second; in the case of the first example, the **bc** prints 2, and in the second prints 5. As you saw above, **bc** also includes the exponentiation operator '^'.

With **bc**, you can also enter numbers with fractional parts. Type the following to illustrate:

```
9.999 * 9.999
```

bc replies:

```
99.980
```

You can save temporary calculations or repeated constants in *variables*. The following example shows you first how to define variables, and second how to use them:

```
a = 1.1
b = 2.2
a
b
a * b
```

Variable names can be longer than one letter.

The basic calculations in the above examples show only part of what **bc** can do. The following section describes simple statements — the assignment of variables and abbreviations — that allow you to perform complex calculations easily.

Simple Statements

Although you can use **bc** as a simple calculator for manipulating numbers, you can take advantage of its greater power by using *variables*. Variables, as noted above, store parts of calculations or constants that you will use repeatedly in calculations. Variable names are simply "words" that you make up. Here are some examples of possible variable names:

```
a
b
totaltaxesdue
ratio
```

To use variables, simply give them a value, use them in a calculation in place of a number, or print them out.

To see how a variable can save you repetitive typing, and protect you from possible errors, invoke **bc** and type the following:

```
x = 9.999
x
x * x
x = x * x
x
```

The following gives the example with **bc**'s replies *in italics*:

```
x = 9.999
x
9.999
x * x
99.980
x = x * x
x
99.980
```

bc did not reply to the assignment statements **x=9.999** and **x=x*x**. However, it did print the value of **x** when requested, and the results of arithmetic using **x**.

Calculations executed with hand-held calculators, with programming languages like C, or with **bc** often use the following formula:

```
x = x + 1
```

To decrease the likelihood of error, **bc** offers you a shorthand expression for this common phrase:

```
x += 1
```

What it means is, "add one to **x**". Type the following example into **bc** to see how this expression works:

```
x = 1
x * x
x += 1
x * x
x += 1
```

Likewise, **bc** provides an abbreviation for:

```
x = x - 2
```

The form should now be familiar:

```
x -= 2
```

The number to the right of the -= or += operator can be replaced with a variable or even another calculation. When you type:

```
i = 4
x = 48
x -= i
x
```

bc in each case replies:

```
44
```

Alternatively, if you type:

```
i = 4
x = 48
x -= i * i
x
```

then **bc** replies:

```
32
```

Similar abbreviations are provided for multiplication, division, remainder, and exponentiation. Here is a summary of this class of operation.

```
a += 2  /* replace a with a plus 2                        */
b += a  /* replace b with b plus a                        */
b -= a  /* replace b with b minus a                       */
c *= b  /* replace c with c multiplied by b               */
c /= a  /* replace c with c divided by a                  */
c %= b  /* replace c with remainder of c divided by b     */
d ^= 3  /* replace d with d raised to the 3rd power       */
```

bc also has an operator that increases a variable by one: '++'. When you type:

```
a = 1
++a
```

then **bc** replies:

```
2
```

To use this operator in an expression, combine it with a variable anywhere that a variable would normally be used. For example, entering

```
b = 1
a = 3
b = ++a
a
b
```

yields:

```
4
4
```

The '++' operator can also be put after a name. The resulting value in the expression is the value of the name *before* it is incremented. However, after the expression is evaluated, the name will have an incremented value. The following example shows the use of '++' both before and after a name:

```
a = 1
b = 1
a++
++b
a
b
```

bc replies:

```
1
2
2
2
```

Operators are used in this manner:

```
a = 1
b = 2
c = a++ + ++b
```

Similar to '++' is '--'. It behaves the same way, except that rather than adding one, it subtracts one.

Numbers with Fractions

Most of the examples presented earlier use whole numbers (integers). However, **bc** can use numbers with fractional parts. This section discusses the use of fractional numbers in **bc** and their precision under different operations.

The Scale of Numbers

The number of digits to the left of the decimal point carried by **bc** depends upon the requirements of the calculation. If you calculate a large number, as in:

```
2^500
```

the result will contain as many digits as needed to express the product.

The number of digits to the right of a decimal point is called the *scale* of the number. Scale depends upon the operation that produces the number of digits, and a variable called **scale** that will be described shortly.

To illustrate simple uses of numbers with fractions, invoke **bc** and then type:

```
a = .01
b = 0.99
a+b
```

bc reply:

```
1.00
```

Addition and Subtraction

bc will dynamically adjust the number of digits in the calculation. It deals similarly with fractional numbers. To the following example

```
a = 0.01
b = 0.001
a + b
```

bc reply:

```
.011
```

In addition and subtraction, the scale of the result is the *larger* of the scales of the two numbers involved. Results are not truncated in addition or subtraction operations.

Scale During Multiplication

Other arithmetic operations act differently with numbers that contain fractions. In the multiplication of two numbers, the scale of the product will at least equal the larger of the scales of the two numbers. For example, the input:

```
1.1 * 1.11
```

results in:

```
1.22
```

Setting the Scale of Results

To increase the number of fractional digits for higher accuracy, **bc** provides the built-in variable **scale**. The following example illustrates the **scale** variable:

```
scale = 3
1.1 * 1.11
```

The result from this example is:

```
1.221
```

Note, however, the scale of the product of a multiplication procedure never exceeds the sum of the scales of the two numbers being multiplied. For example,

```
scale = 10
1.1 * 1.11
```

yields the result:

```
1.221
```

If the variable **scale** is less than the sum of the scales of the numbers being multiplied, then the product will have a scale equal to that of the variable **scale**. For example,

```
scale = 4
1.11 * 2.222
```

yields:

```
2.4664
```

The scales of the operands are 2 and 3. The larger scale is 3, so the result of a multiplication will have a scale of at least 3, no matter what **scale** is set to. Also, the sum of the scales is 5, so the result will never have more than 5 digits to the right of the decimal point. In this example, **scale** has been set to a number between 3 and 5, namely 4. Therefore, the result has a scale of 4.

Scale for Divisions

For division and remainder, the scale of the result is determined only by the value of the variable **scale**. For example,

```
scale = 13
14 / 13
14 % 13
```

yields:

```
1.0769230769230
.0000000000010
```

For non-whole numbers, as well as for integers, the definition of remainder is chosen so that the relationship

```
dividend = (divisor * quotient) + remainder
```

is true.

Scale From Exponentiation

bc sets the **scale** of a result of exponentiation as if repeated multiplications had been performed. Thus, for

```
5.992 ^ 5
```

the scale is chosen as if you typed:

```
n = 5.992
n * n * n * n * n
```

That is, the default is the scale of the largest (or, in this case, the only) number being multiplied; and scale cannot exceed the sum of the numbers being multiplied. Thus, the scale of the product in this example has a default setting of 3, and can be reset up to 15.

What Is the Current Scale?

The variable **scale** is just like other variables: you can assign values to it, as above. Because it is like regular variables, you can also use it in operations, as in this example:

```
scale += 1
```

You can also print its value:

```
scale
```

The value of the **scale** variable is zero until you explicitly change it.

The if Statement

The statements shown so far have been either assignment statements, giving a new value to a variable; or an expression, which prints the resulting value. Several other kinds of statements are available. These give you power to write programs that make decisions and perform iterative computations.

Using the if Statement

To see the **if** statement in action, type the following example into **bc**:

```
x = 3
if (x < 5) x
if (x > 5) -x
```

The reply is:

```
3
```

If the input is:

```
x = 6
if (x < 5) x
if (x > 5) -x
<return>
```

bc replies:

```
-6
```

The part of the **if** statement in parentheses, such as **(x > 5)**, determines whether **bc** executes the statement that follows it, such as **-x**. If the expression is false, the following statement is not executed. If the expression is true, the following statement is executed.

Comparisons

The decision expression in an **if** statement is enclosed in parentheses. The decision can be based upon a comparison of two operands, or numbers. The kinds of comparisons that can be done are:

==	First operand equal to second
!=	First operand not equal to second
<=	First operand less than or equal to second
<	First operand less than second
>=	First operand greater than or equal to second
>	First operand greater than second

The **if** statement can include the sorts of the simple statements already shown. You can also include an **if** statement, as well as the **while**, **do**, and **for** statements, which will be discussed below. The following example illustrates the use of an **if** statement within an **if** statement:

```
a = 2
b = 6
if (a >= 2) if (b > a) a + b
<return>
```

bc replies, simply:

```
8
```

Because both of the **if** conditions were true, **bc** proceeded to add **a** and **b**.

Grouped Statements

You can place more than one statement after the expression part of the **if** statement by using grouping braces '{' and '}'. This can be useful if you want to perform several calculations based on the result of an **if** statement comparison. The following example prints the value of **a** and **b** if the value of **b** is less than the value of **a**:

```
a = 1
b = .99
if (a > b) {
    a
    b
}
```

bc replies:

```
1
.99
```

Any statement may be enclosed within the group braces, as the following example shows:

```
a = 1
b = .99
if (a > b) {
    a
    b
    if ((a + b) >= 2) a + b
}
```

Many Statements Per Line

To this point, all of our examples typed each statement on its own line. This includes the group braces '{' and '}', the latter of which must appear on a line by itself. You can, however, place several statements on one line if you separate them with semicolons. If you do this, remember that the semicolon rather than the carriage return separates the statements. For example, if you type:

```
a = 1;b = 2;c = 3
a;b;c
```

bc replies:

```
1
2
3
```

You can use this in combination with the group braces:

```
a = 1;b = 2;c = 3
if ((a +b) >= c) {
    a; b; c; a + b; }
```

The reply from **bc** is:

```
1
2
3
3
```

This example can be compressed even further by putting all of the **if** statement on one line:

```
a = 1;b = 2;c = 3
if ((a + b) >= c) { a; b; c; a + b; }
```

You do not need to follow the '}' with a semicolon.

The while Statement

The **while** statement repeats calculations. This is useful in successive approximation calculations. The following example of the **while** loop prints the numbers one through ten:

```
i = 1
while (i <= 10) {
      i
      i = i + 1
}
```

bc replies:

```
1
2
3
4
5
6
7
8
9
10
```

The statement

```
i = i + 1
```

adds 1 to the variable **i**. The expression

```
(i <= 10)
```

compares **i** with 10. While **i** is less than or equal to 10, the **while** loop executes. When **i** is

increased to greater than 10, the loop stops executing.

bc checks the comparison expression for the **while** loop before the loop is entered for the first time. If the comparison fails, the loop is not executed at all; otherwise the processing repeats as long as the comparison is true. For example, the following statements do not print anything:

```
i = 0
while (i > 1) i
quit
```

Abbreviations in the while Statement

If we recall the assignment statements from the previous section, we can shorten the **while** counting-to-ten example to:

```
i = 1
while (i <= 10) {
        i
        i += 1
}
```

The result remains the same — a list of numbers from one to ten.

Another abbreviation of the example uses the '++' operator. The variable **i** is incremented, then tested in the **while** expression, which simplifies the entire example to:

```
i = 0
while (++i <= 10) i
```

Before the **while** is executed, **i** is set to zero. Then, the **while** expression increments the value of **i** before it is used or compared, Thus, the first value compared, then printed, is one.

Finally, the example calculation can be shortened to one line. If a variable in **bc** is used before it is initialized, it will have the value of zero. For example:

```
zip
```

prints:

```
0
```

Using this in our counting-to-ten example yields:

```
while (++i <= 10) i
```

The for Statement

for is a statement that controls the execution of other **bc** statements. You should use **for** to write a formula to control the number of times a value is computed.

The previous section demonstrated how to print the numbers one to ten using a **while** statement. The following does the same task with a **for** statement:

```
for (i=1; i <= 10; ++i) i
```

Three Parts of the for Statement

The **for** statement is more complex than the **while** statement; its controlling expressions have three parts.

The first part, shown here in italics

```
for (i=1; i <= 10; ++i) i
```

sets up the initial condition. The second part

```
for (i=1; i <= 10; ++i) i
```

tests whether more iterations should be performed. **bc** performs this test *before* it executes the statements that are subordinate to the **for** statement. If the test fails, no more iterations are performed.

The third part

```
for (i=1; i <= 10; ++i) i
```

is performed at the end of each iteration. In practically every instance, this part of the **for** statement modifies the value of the variable that the second part tests.

Taken together, these statements (1) set **i** to zero; (2) check whether **i** is less than or equal to ten; (3) if **i** proves to be so, prints **i**, and then increases it by one.

The following example of the **for** statement adds the squares of the numbers one through ten, prints each square, and then prints the sum of the squares at the end.

```
sum = 0
for (n=1; n <= 10; ++n) {
    sq = n * n
    sq
    sum += sq
}
sum
```

The result is:

```
1
4
9
16
25
36
49
64
81
100
385
```

Similarities Between the for and while Statements

To illustrate the similarity between the **for** statement and the simpler **while** statement, the following rewrites the above example, substituting the **while** for the **for**:

```
sum = 0
n = 1
while (n <= 10) {
        sq = n * n
        sq
        sum += sq
        ++n
}
sum
```

You should notice one difference when you enter this example. In the **while** version of the example, the

```
++n
```

prints out the new value of **n**, whereas in the **for** example, the value is not printed.

Functions in bc

bc allows you to name routines that you use repeatedly. You can then call them by name without having to retype them; obviously, this can be a great time-saver. These named routines are called *functions*. This section shows you how to define and use functions for your **bc** calculations.

Example of Function Use

The following example defines a function that calculates the area of a circle from its radius.

```
scale = 5
pi = 3.14159
define area (radius) {
        r2 = radius * radius
        return (pi * r2);
}
area (1.00);
area (2.00);
area (56);
```

The results will be:

```
3.14159
12.56636
9852.02624
```

The **define** keyword tells **bc** that you are defining a function. The name of the function follows. Then, in parentheses, come the *parameters* of the function. In this example, the only parameter, or *argument*, of the function is **radius**. Most functions have arguments, but they are not mandatory.

The **return** statement defines the value of the function. In the area example, the expression:

```
area (1.00)
```

references the function **area**. **bc** then performs the calculation described by your definition of the function area. The number

```
1.00
```

is substituted wherever the parameter **radius** is shown.

The statement

```
r2 = radius * radius
```

is then executed, yielding this result:

```
1.00
```

Then, the statement

```
return (pi * r2)
```

calculates the area and returns its value. The statement

```
area (1.00)
```

then has the value calculated in the return statement.

Functions Using Other Functions

Functions in **bc** perform calculations using the same expressions as the rest of the **bc** program. This includes the use of functions. The **area** program can be written using another function, **sq**, to calculate the square of a number:

```
scale = 5
pi = 3.14159
define sq (number) {
       return (number * number)
       }
define area (radius) {
       return (sq (radius) * pi)
       }
area (1.00);
area (2.00);
area (56);
```

Again, the results will be identical:

```
3.14159
12.56636
9852.02624
```

Functions That Call Themselves

Not only can functions call other functions and perform regular calculations; a function can use itself in calculations. An example of this is the Fibonacci calculation:

```
define fib (f) {
        if (f==0) return (0)
        if (f==1) return (1)
        if (f > 1) return (fib (f-1) + fib (f-2))
}
fib (5)
fib (20)
```

Fibonacci numbers are defined in the following way: Fibonacci number zero is zero; similarly, Fibonacci number one is one. Any other Fibonacci number is defined as the sum of the two previous Fibonacci numbers. Fibonacci numbers are defined only for non-negative integers.

The defined function **fib** follows this definition by returning zero if the number requested is zero and one if the argument is one. If the number is neither of these, then the function calls itself to calculate the previous two numbers of the series and adds them together.

The auto Statement

Many functions that call other functions, including themselves, may require variables that are not changeable by the rest of the program. This is signalled to **bc** by the **auto** statement:

```
auto var1, var2
```

This declares **var1** and **var2** as local to the function that contains them.

To illustrate the use of **auto**, the following **bc** program calculates the factorial of a number:

```
define factorial (number) {
        auto value, i
        value = 1
        for (i = 1; i <= number; ++i) value *= i
        return (value)
}
value = 3
factorial (value)
i = 99
factorial (20)
value
i
```

The result is:

```
6
2432902008176640000
3
99
```

The first number, 6, results from:

```
factorial (value)
```

The second number is from:

```
factorial (20)
```

The last two numbers are from **value** and **i**, and are included to demonstrate that the variables in the function **factorial** appearing in this statement:

```
auto value, i
```

are separate from the variables of the same name in the rest of the program.

If the function calls itself, as the **fib** example does above, any variable names noted in the **auto** statement are handled separately for each call of the function.

Programs in a File

Because its programs can be quite complex, **bc** lets you keep them in files. This lets you build a library of **bc** programs and functions that can be called up easily.

Using a Program From a File

To illustrate the use of programs stored in a file, type the following example into file **fib.bc** COHERENT using the editor of your choice. The program defines the function **fib**:

```
define fib (f) {
        if (f==0) return (0)
        if (f==1) return (1)
        if (f > 1) return (fib (f-1) + fib (f-2))
}
```

To use a **bc** program that has been stored in a file, enter the file name on the **bc** command line, like this:

```
bc fib.bc
```

The function definition will be read in by **bc** and ready for your use. To use the function, simply type the function name with parameters.

So, if you type:

```
bc fib.bc
fib (6)
quit
```

bc will reply:

```
8
```

Using Libraries

You can enter several useful programs in their own files and call them into **bc** at the same time. The following example creates another function that calculates the sum of the squares of integers up to a given number. Enter it into COHERENT, and name it **sumsq.bc**:

```
define sumsq (number) {
        auto i, sum
        sum = 0
        for (i = number; i > 0; --i) sum += i ^ 2
        return (sum)
}
```

Now, you can use the **sumsq** function to print the sum of the squares for each number from one to ten:

```
bc sumsq.bc
for (i = 1; i <= 10; ++i) sumsq (i)
quit
```

The result is:

```
1
5
14
30
55
91
140
204
285
385
quit
```

You can use the two functions stored in a file to print the difference between the sum of the squares of numbers, and the Fibonacci number:

```
bc fib.bc sumsq.bc
for (i = 1; i <= 10; ++i) sumsq (i) - fib (i)
quit
```

The result of this questionable computation is:

```
0
4
12
27
50
83
127
183
251
330
```

The bc Library

COHERENT provides an extended library to go with **bc**. It includes the following functions:

atan(*z*)	arctangent of *z*
cos(*z*)	cosine of *z*
exp(*z*)	exponential function of *z*
j(*n*,*z*)	*n*th order Bessel function of *z*
ln(*z*)	natural logarithm of *z*
pi	the value of pi to 100 digits
sin(*z*)	sine of *z*

The library is stored in file **/usr/lib/lib.b**. To use the library, invoke the **bc** command with the **-l** option.

To show how the library can be used in your work the following example computes the sine of an angle of one-third radian with scale set to 20:

```
bc -l
scale = 20
sin (1/3)
quit
```

The result is:

```
.32719469679615224418
```

Summary

The Lexicon entry for **bc** summarizes its commands, features, and libraries. It will also you refer you to related commands and functions.

Section 5:

The C Language

C is a computer language invented by Dennis Ritchie and Ken Thompson at AT&T Bell Laboratories in the early 1970s. In the approximately 20 years since its creation, C has become one of the most popular compter languages in the world. C is powerful, flexible; it is highly portable, and has been implemented on practically every computer, and under practically every operating system, in the world.

C is the "native language" of the COHERENT system. COHERENT is written in C, and it includes a powerful C compiler among its suite of language tools for your use. You do not need to know C to use COHERENT to great advantage; however, if you plan to program under COHERENT, you would be well advised to become at least passably acquainted with it.

This tutorial is an introduction to the COHERENT C compiler and to the C language itself. The first part of this section describes how to compile programs under COHERENT. The second part is a brief tutorial in the C language itself.

Compiling C Programs under COHERENT

A C compiler is a program that transforms files of C source code into machine code. Compilation is a complex process that involves several steps; however, COHERENT simplifies it with the command **cc**, which controls all the actions of the compiler.

Try the Compiler

Before we launch into a lengthy explanation of what **cc** is and what it does, you can get a feel for it by trying it with a simple example. To begin, type the following to create a simple C program:

```
cat >hello.c
main() {
        printf("Hello, world\n");
}
<ctrl-D>
```

This creates a simple C program called **hello.c**. Now, compile your program by typing the following command:

```
cc -V hello.c
```

If you typed the program correctly, **cc** will print something like the following on your screen:

```
/lib/cc0 D23400000100 hello.c /tmp/cc15029b
/lib/cc1 D23000000100 /tmp/cc15029b /tmp/cc15029a
/lib/cc2 D23000000100 /tmp/cc15029a hello.o /tmp/cc15029b
rm /tmp/cc15029a
rm /tmp/cc15029b
/bin/ld -X /lib/crts0.o hello.o /lib/libc.a
rm hello.o
```

What each of these messages means will be described below. If you receive an error message, try re-typing the program, and then re-compile it. When compilation is successfully completed, you will now have an executable program called **hello**. To invoke it, type:

```
hello
```

It should print the following on your screen:

```
Hello, world
```

As you can see, **cc** makes it easy to transform a file of C code into an executable program.

Phases of Compilation

As you noticed, **cc** printed a number of messages on your screen as it compiled **hello.c**. The reason you saw the messages was that compilation was performed with the **-V** option to **cc**; this tells **cc** to print a verbose output that describes each of its actions. **cc** prints numerous messages because the COHERENT C compiler is not just one program, but a number of different programs that work together. Each program performs a *phase* of compilation. The following summarizes each phase:

cpp The C preprocessor. This processes any of the '#' directives, such as **#include** or **#ifdef**, and expands macros.

cc0 The parser. This phase parses programs. It translates the program into a parse-tree format, which is independent of both the language of the source code and the microprocessor for which code will be generated.

cc1 The code generator. This phase reads the parse tree generated by **cc0** and translates it into machine code. The code generation is table driven, with entries for each operator and addressing mode.

cc2 The optimizer/object generator. This phase optimizes the generated code and writes the object module.

cc3 COHERENT also includes a fifth phase, called **cc3**, which can be run after the object generator, **cc2**. **cc3** generates a file of assembly language instead of a relocatable object module. **cc3** allows you to examine the code generated by the compiler. You did not see this phase when you compiled **hello.c** because this phase is optional and you did not request it. If you want COHERENT to generate assembly language, use the **-S** option on the **cc** command line.

Unless you specify the **-S** option, **cc** creates an *object module* that is named after the source file being compiled. This module has the suffix **.o**. An object module is *not* executable; it contains only the code generated by compiling a C source file, plus information needed to link the module with other program modules and with the library functions.

As the final step in its execution, **cc** calls the linker **ld** to produce an executable program.

As you can see, **cc** also removes the temporary files it creates to pass information from one compiler phase to another. If your program is built out of only one file of C source code, it also deletes the object module that it creates after that module is linked to create an executable program.

Renaming Executable Files

When **cc** compiles a source file, by default it names the executable program after the *first* source file named on the **cc** command line. If you wish, you can give the executable file a different name. Use the **-o** (output) option, followed by the desired name.

Floating-Point Numbers

Often, you will need to use floating-point numbers in your programs. If you are unsure what a floating-point number is, see the Lexicon entry for **float**.

The routines that print floating-point numbers are large, and most C programs do not need to print floating-point numbers; therefore, the code to perform floating-point arithmetic is not included in a program by default. You must ask **cc** to include these routines with your program by using the **-f** option to **cc**.

To see how this works, let's modify **hello.c** to use floating-point numbers. Edit **hello.c** by typing the following commands:

```
ed hello.c
2
c
        printf("Hello, world %f\n", 123.4);
.
w
q
```

Now, compile the program with the same command line as before:

```
cc -V hello.c
```

When compilation has finished, type **hello**. You'll see the following output:

```
You must compile with the -f flag
to include printf() floating point.

Hello, world
```

COHERENT is telling you that you are using a floating-point number but that you did not compile the program to include code to process floating-point numbers. Now, recompile the program using the **-f** option to **cc**:

```
cc -V -f hello.c
```

When compilation has finished, type **hello**. If you typed the program correctly, you will see the following:

```
Hello, world 123.400000
```

As you can see, **hello** is now displaying the floating-point number **123.4** for you. For detailed information on **printf**, see its entry in the Lexicon; **printf** is also discussed in the

tutorial section below.

Compiling Multiple Source Files

Many programs are built from more than one file of C source code. For example, the program **factor**, which is provided with COHERENT, is built from the C source files **factor.c** and **atod.c**. To produce the executable program **factor**, both source files must be compiled; the linker **ld** then joins them to form an executable file.

To compile a program that uses more than one source file, type all of the source files onto the **cc** command line. For example, to compile **factor** you would type the following:

```
cc -o factor -f factor.c atod.c -lm
```

This command compiles both C source files to create the program **factor**.

In the above example, **cc** produces the non-executable object modules **factor.o** and **atod.o**, and then links them to produce the executable file **factor**.

The argument **-lm** tells **cc** to include routines from the mathematics library when the object modules are linked. This option must come *after* the names of all of the source files, or the program will not be linked correctly.

Linking Without Compiling

When you are writing a program that consists of several source files, you will need to compile the program, test it, and then change one or more of the source files. Rather than recompile all of the source files, you can save time by recompiling only the modified files and relinking the program.

For example, if you modify the **factor** program by changing the source file **factor.c**, you can recompile **factor.c** *and* relink the entire program with the following command:

```
cc -o factor -f factor.c atod.o -lm
```

This **cc** command refers to the C source file **factor.c** and the *object module* **atod.o**. **cc** recognizes that **atod.o** is an object module and simply passes it to the linker **ld** without recompiling it. You will find this particularly useful when your programs consist of many source files and you need to compile only a few of them.

To simplify compiling, especially if you are developing systems that use many source modules, you should consider using the **make** utility that is included with COHERENT. For more information on **make**, see its entry in the Lexicon, or see the tutorial for **make** that appears later in this manual.

Compiling Without Linking

At times, you will need to compile a source file but not link the resulting object module to the other object modules. You will do this, for example, to compile a module that you wish to insert into a library. Use **cc**'s option **-c** to tell **cc** not to link the compiled program. This option is often used to create relocatable object modules that can be archived into a library for later use.

For example, if you wanted just to compile **factor.c** without linking it, you would type:

```
cc -c factor.c
```

To link the resulting object module with the object module **atod.o** and with the appropriate libraries, type the following command:

```
cc -o factor -f factor.o atod.o -lm
```

Assembly-Language Files

C makes most assembly language programming unnecessary. However, you may wish to write small parts of your programs in assembly language for greater speed or to access processor features that C cannot use directly. COHERENT includes an assembler, named **as**, which is described in detail in the Lexicon.

To compile a program that consists of the C source file **example.c** and the assembly-language source file **example.s**, simply use the **cc** command as usual:

```
cc -o example example1.c example2.s
```

cc recognizes that the suffix **.s** indicates an assembly-language source file, and assembles it with **as**; then it links both object modules to produce an executable file.

Changing the Size of the Stack

The *stack* is the segment of memory that holds function arguments, local variables, and function return addresses. COHERENT by default sets the size of the stack to two kilobytes (2,048 bytes). This is enough stack space for most programs; however, some programs, such as the example program on page 26 of the first edition of *The C Programming Language*, require more than two kilobytes of stack. A program that uses more than its allotted amount of stack will cause a *stack overflow*, which will cause your program to crash.

The size of the stack cannot be altered while a program is running. Should your program need more than two kilobytes of stack, use the COHERENT command **fixstack**. For more information, see the entry for **fixstack** in the Lexicon.

Where To Go From Here

This discussion of the **cc** command is by no means complete, but it includes enough information for you to begin to compile your programs. The Lexicon's entry for **cc** gives all of the command-line options available with **cc**. The Lexicon also has entries for **cpp**, the compiler phases, and for the linker **ld**, and describes them at greater length. All error messages generated by **cc** and by the assembler **as** appear in the appendix to this manual.

The next section in this tutorial introduces the C programming language.

C for Beginners

This section briefly introduces the C programming language. It is in two parts. The first part describes what a programming language is, and gives the history of the C programming language. This section also introduces some concepts basic to C, such as *structured programming*, *pointer*, and *operator*. The second part walks through a C programming session. It emphasizes how a C programmer deals with a real problem, and demonstrates some aspects of the language.

This chapter is not designed to teach you the entire C language. It introduces you to C, so you can read the rest of this manual with some understanding. We urge you to look up individual topics of C programming in the Lexicon, and especially to study the example programs given there.

Programming Languages and C

Before beginning with C, it is worthwhile to review how a microprocessor and a computer language work.

A *microprocessor* is the part of your computer that actually computes. Built into it is a group of *instructions*. Each instruction tells the microprocessor to perform a task; for example, one instruction adds two numbers together, another stores the result of an arithmetic operation in memory, and a third copies data from one point in memory to another.

Together, a microprocessor's instructions form its *instruction set*. The instruction set is, in effect, the microprocessor's "native language".

A microprocessor also contains areas of very fast storage, called *registers*. The registers are essential to arithmetic and data handling within the microprocessor. How many registers a microprocessor has, and how they are designed, help to determine how much memory the microprocessor can read and write, or *address*, and how the microprocessor handles data.

A *computer language*, as the name implies, lets a human being use the microprocessor's instruction set. The lowest level language is called "assembly language". In assembly language, the programmer calls instructions directly from the microcomputer's instruction set, and manipulates the registers within the microprocessor. To write programs in assembly language, a programmer must know both the microprocessor's instruction set and the configuration of its registers.

Assembly and High-Level Languages

With assembly language, the programmer can tailor the program specifically to the microprocessor. However, because each microprocessor has a unique instruction set and configuration of registers, a program written in one microprocessor's assembly language cannot be run on another microprocessor. For example, no program written in the assembly language for the Motorola 68000 microprocessor can be run on the IBM PC or any PC-compatible computer. The program must be entirely rewritten in the assembly language for the Intel i8086 microprocessor, which is difficult and time consuming.

A *high-level language* helps programmers to avoid these problems. The programmer does not need to know the microprocessor in detail; instead of specific microprocessor instructions, he writes a set of logical constructions. These constructions are then handed to another program, which translates them into the instructions and registers calls used by a specific microprocessor. In theory, a program written in a high-level language can be run on any microprocessor for which someone has written a translation program.

A high-level language allows the programmer to concentrate on the task being executed, rather than on the details of registers and instructions. This means that programs can be written more quickly than in assembly language, and can be maintained more easily.

So, What Is C?

As noted earlier, C was invented at AT&T Bell Laboratories by Dennis Ritchie and Ken Thompson. They created C specifically to re-write the UNIX operating system from PDP-11 assembly language. Ritchie designed C to have the power, speed, and flexibility of assembly language, but the portability of high-level languages.

In 1978, Ritchie and Brian W. Kernighan published *The C Programming Language*, which describes and defines the C language. *The C Programming Language* is the "bible" of C, a standard work to which all programmers can refer when writing their programs.

Because C is modeled after assembly language, it has been called a "medium-level" language. The programmer doesn't have to worry about specific registers or specific instructions, but he can use all of the power of the computer almost as directly as he can with assembly language.

Because C was written by experienced programmers for experienced programmers, it makes little effort to protect a programmer from himself. A programmer can easily write a C program that is legal and compiles correctly but crashes the program. Also, C's punctuation marks, or "operators", closely resemble each other. Thus, a mistake in typing can create a legal program that compiles correctly but behaves very differently from what you expect.

Structured Programming

C is a *structured language*. This means that a C program is assembled from a number of sub-programs, or *functions*, each of which performs a discrete task. If this concept is difficult to grasp, consider the following example.

Suppose you want to turn a file of text into upper-case letters and print it on the screen. This job seems simple, but a program to do it must perform five tasks:

1. Read the name of the file to open.
2. Open the file so it can be read, in much the same way that you must open a book before you can read it.
3. Read the text from the file.
4. Turn what is read into upper-case letters.
5. Print the transformed text onto the screen.

A good program will also perform the following tasks:

1. Check that the file requested actually exists.
2. Check that the file requested is actually a text file rather than a file of binary information; the latter makes very little sense when printed on the screen.
3. Close the program neatly when the work is finished.
4. Stop processing and print an error message if a problem occurs.

A structured language like C allows you to write a separate function for each of these tasks.

A structured programming language offers two major advantages over a non-structured language. First, it is easier to debug a function than an entire program because the function can be unplugged from the program as a whole, made to work correctly, and then plugged back in again. Second, once a function works, it can be used again and again in different programs. This allows you to create a *library* of reliable functions that you can pull off the shelf whenever you need them.

The functions within a program communicate by passing values to each other. The value being passed can be an integer, a character, or — most commonly — an address within memory where a function can find data to manipulate. This passing of addresses, or *pointers*, is the most efficient way to manipulate data because by receiving one number, a function can find its way to a large amount of data. This speeds up a program's execution.

C adds some extra tools to help you construct programs. To begin, C allows you to store functions in compiled form. These precompiled functions are added only when the program is finally loaded into memory; this spares you the trouble of having to recompile the same code again and again. Second, C adds a preprocessor that expands definitions, or *macros*, and pulls in special material stored in *header files*. This allows you to store often-used definitions in one file and use them just by adding one line to your program.

Writing a C Program

As noted above, a C program consists of a bundle of sub-programs, or *functions*, which link together to perform the task you want done. Every C program must have one function that is called **main**. This is the main function; when the computer reads this, it knows that it must begin to execute the program. All other functions are subordinate to **main**. When the **main** function is finished, the program is over.

To see how these elements work, review the program **hello.c**, which you worked with earlier in this tutorial:

```
main()
{
        printf("Hello, world\n");
}
```

As you can see, this program begins with the word **main**. The program begins to work at this point. The parentheses after **main** enclose all of the *arguments* to **main** — or would, if this program's **main** took any. An argument is an item of information that a function uses in its work.

The braces '{' and '}' enclose all the material that is subsidiary to **main**.

The word "printf" *calls* a function called **printf**. This function performs formatted printing. The line of characters (or "string") *Hello, world* is the argument to **printf**: this argument is what **printf** is to print.

The characters '\n' stand for a newline character. This character "tosses the carriage", or moves the cursor to a new line and returns it to the leftmost column on your screen. Using this character ensures that when printing is finished, the cursor is not left fixed in the middle of the screen. Finally, the semicolon ';' at the end of the command indicates that the function call is finished.

One point to remember is that **printf** is *not* part of the C language. Rather, it is a *function* that was written by Mark Williams Company, then compiled and stored in a library for your use. This means that you do not have to re-invent a formatted printing function to perform this simple task: all you have to do is *call* the one that Mark Williams has written for you.

Although most C programs are more complicated than this example, every C program has the same elements: a function called **main**, which marks where exection begins and ends; braces that fence off blocks of code; functions that are called from libraries; and data passed to functions in the form of arguments.

A Sample C Programming Session

This section walks you through a C programming session. It shows how you can go about planning and writing a program in C.

C allows you to be precise in your programming, which should make you a stronger programmer. Be careful, however, because C does exactly what you tell it to do, nothing more and nothing less. If you make a mistake, you can produce a legal C program that does very unexpected things.

Designing a Program

Most programmers prefer to work on a program that does something fun or useful. Therefore, we will write something useful: a version of the COHERENT utility **scat**, that we'll call **display**. It will do the following:

1. Open a text file on disk.
2. Display its contents in 23-line chunks (one full screen).
3. After displaying a chunk, wait to see if the user wants to see another chunk. If the user presses the **<return>** key alone, display another chunk; if the user types any other key before pressing the **<return>** key, exit.
4. Exit automatically when the end of file is reached.

As you can see, the first step in writing a program is to write down what the program is to do, in as much detail as you can manage, and preferably in complete sentences.

Now, invoke **ed** or MicroEMACS and get ready to type in the program:

```
ed display.c
```

or:

```
me display.c
```

We suggest that you use the MicroEMACS editor, because this tutorial will make numerous changes to the program as it progresses and it will be easier to see these changes in context if you use a screen editor rather than a line editor. The rest of this tutorial assumes that you are using MicroEMACS. If you are not familiar with MicroEMACS, it is briefly described in *Using the COHERENT System*. A tutorial for MicroEMACS also appears in this manual, or you may wish to see the entry for **me** in the Lexicon.

In the above commands, the suffix **.c** on the file name indicates that this is a file of C code. If you do not use this suffix, the **cc** command will not recognize that this is a file of C code and will refuse to compile it.

Begin by inserting a description of the program into the top of the file in the form of a *comment*. When a C compiler sees the symbol '/*', it throws away everything it reads until it sees the symbol '*/'. This lets you insert text into your program to explain what the program does.

Type the following:

```
/*
 * Truncated version of the 'scat' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <return>, print another 23 lines.
 * If user types any other key, exit.
 * Exit when EOF is read.
 */
```

Save what you have typed by pressing **<ctrl-X>** and then **<ctrl-S>**. Now, anyone, including you, who looks at this program will know exactly what it is meant to do.

The main() Function

As described earlier, the C language permits *structured programming*. This means that you can break your program into a group of discrete functions, each of which performs one task. Each function can be perfected by itself, and then used again and again when you need to execute its task. C requires, however, that you signal which function is the *main* function, the one that controls the operation of the other functions. Thus, each C program must have a function called **main()**.

Now, add **main()** to your program. Type the code that is shaded, below:

```
/*
 * Truncated version of the 'scat' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <return>, print another 23 lines.
 * If user types any other key, exit.
 * Exit when EOF is read.
 */

main()
{
}
```

The parentheses "()" show that **main** is a function. If **main** were to take any arguments, they would be named between the parentheses. The braces "{}" delimit all code that is subordinate to **main**; this will be explained in more detail below.

Note that the shortest legal C program is **main(){}**. This program doesn't do anything when you run it, but it will compile correctly and generate an executable file.

Now, try compiling the program. Save your text by typing **<ctrl-X><ctrl-S>**, and then exit from the editor by typing **<ctrl-X><ctrl-C>**. Compile the program by typing:

```
cc display.c
```

When compilation is finished, type **display**. The shell will pause briefly, then return the prompt to your screen. As you can see, you now have a legal, compilable C program, but one that does nothing.

Open a File and Show Text

The next step is to install routines that open a file and print its contents. For the moment, the program will read only a file called **tester**, and not break it into 23-line portions.

Type the shaded lines into your program, as follows:

```
/*
 * Truncated version of the 'scat' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <return>, print another 23 lines.
 * If user types any other key, exit.
 * Exit when EOF is read.
 */

#include <stdio.h>

main()
```

```
{
        char string[128];
        FILE *fileptr;

/* Open file */
        fileptr = fopen("tester", "r");

/* Read material and display it */
        for (;;) {
                fgets(string, 128, fileptr);
                printf("%s", string);
        }
}
```

Note first how comments are inserted into the text, to guide the reader.

Now, note the lines

```
        char string[128];
        FILE *fileptr;
```

These *declare* two data structures. That is, they tell COHERENT to set aside a specific amount of memory for them.

The first declaration, **char string[128];**, declares an array of 128 **char**s. A **char** is a data entity that is exactly one byte long; this is enough space to store exactly one alphanumeric character in memory, hence its name. An *array* is a set of data elements that are recorded together in memory. In this instance, the declaration sets aside 128 **char**s-worth of memory. This declaration reserves space in memory to hold the data that your program reads.

The second declaration, **FILE *fileptr**, declares a *pointer* to a **FILE** structure. The asterisk shows that the data element points to something, rather than being the thing itself. When a variable is declared to be a pointer, the C compiler sets aside enough space in memory to hold an *address*. When your program reads that address, it then knows where the actual data are residing, and looks for them there. C uses pointers extensively, because it is much more efficient to pass the address of data than to pass the data themselves. You may find the concept of pointers to be a little difficult to grasp; however, as you gain experience with C, you will find that they become easy to use.

The **FILE** structure is the data entity that holds all the information your program needs to read information from or write information to a file on the disk. For a detailed discussion of the **FILE** structure, see its entry in the Lexicon. For now, all you need to remember is that this declaration sets aside a place to hold a pointer to such a structure, and the structure itself holds all of the information your program needs to manipulate a file on disk. In effect, the variable **fileptr** is used within your program as a synonym for the file itself.

Now, the line

```
        fileptr = fopen("tester", "r");
```

opens the file to be read. The function **fopen** *opens* the file, fills the **FILE** structure, and fills the variable **fileptr** with the address of where that structure resides in memory.

fopen takes two arguments. The first is the name of the file to be opened, within quotation marks. The second argument indicates the *mode* in which to open the file; **r** indicates that the file will be read rather than written into.

The lines

```
for(;;)
{
```

begin a *loop*. A loop is a section of code that is executed repeatedly until a condition that you set is fulfilled. For example, you may define a loop that executes until the value of a particular variable becomes greater than zero.

for is built into the C language. Note that it has braces, just like **main()** does; these braces mean that the following lines, up to the next right brace (}) are part of this loop. You can set conditions that control how a **for** loop operates; in its present form, it will loop forever. This will be explained in more detail shortly.

Two library functions are executed within the loop. The first,

```
fgets(string, 128, fileptr);
```

reads a line from the file named in the **fileptr** variable, and writes it into the character array called **string**. The middle argument ensures that no more than 128 characters will be read at a time. The second line within this loop,

```
printf("%s", string);
```

prints the line. **printf** is a powerful and subtle function; in its present form, it prints on the screen the string contained in the variable *string*.

Finally, the line at the top of the program:

```
#include <stdio.h>
```

tells C preprocessor **cpp** to read the *header file* called **stdio.h**. The term "STDIO" stands for "standard input and output"; **stdio.h** declares and defines a number of routines that will be used to read data from a file and write them onto the screen.

When you have finished typing in this code, again compile the program as you did earlier. If an error occurs, check what you have typed and make sure that it *exactly* matches the code shown on the previous page. If you find any errors, fix them and then recompile. If errors persist, check it in the table of error messages that appear at the end of this tutorial.

When compilation is finished, execute **display** as you did earlier. You will see the text from **display.c** scroll across the screen. When the text is finished, however, the COHERENT prompt does not return; you have not yet inserted code that tells the program to recognize that the file is finished. Type **<ctrl-C>** to break the program and return to COHERENT

Accepting File Names

Of course, you will want **display** to be able to display the contents of any file, not just files named **display.c**. The next step is to add code that lets you pass arguments to the program through its command line. This task requires that you give the **main()** function two arguments. By tradition, these are always called **argc** and **argv**. How they work will be described in a moment.

The enhanced program appears as follows. You should change or insert the lines that are shaded:

```
/*
 * Truncated version of the 'scat' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <return>, print another 23 lines.
```

```
 * If user types any other key, exit.
 * Exit when EOF is read.
 */

#include <stdio.h>
#define MAXCHAR 128

main(argc, argv)
/* Declare arguments to main() */
int argc;
char *argv[];
{
        char string[MAXCHAR];
        FILE *fileptr;

/* Open file */
        fileptr = fopen(argv[1], "r");

/* Read material and display it */
        for (;;)
        {
                fgets(string, MAXCHAR, fileptr);
                printf("%s", string);
        }
}
```

First, a small change has been added: the line

```
#define MAXCHAR 128
```

defines the *manifest constant* **MAXCHAR** to be equivalent to 128. This is done because the "magic number" 128 is used throughout the program. If you decide to change the number of characters that this program can handle at once, all you would have to do is to change this one line to alter the entire program. This cuts down on mistakes in altering and updating the program. If you look lower in the program, you will see that the declaration

```
char string[128]
```

has been changed to read

```
char string[MAXCHAR]
```

The two forms are equivalent; the only difference is that the latter is easier to use. It is a good idea to use manifest constants wherever possible, to streamline changes to your program.

Now, look at the line that declares **main()**. You will see that **main()** now has two arguments: **argc** and **argv**.

The first is an **int**, or integer, as shown by its declaration — **int argc;**. **argc** gives the *number* of entries typed on a command line. For example, when you typed

```
display filename
```

the value of **argc** was set to two: one for the command name itself, and one for the file-name argument. **argc** and its value are set by the compiler. You do not have to do anything to ensure that this value is set correctly.

argv, on the other hand, is an array of pointers to the command line's arguments. In this instance, **argv[1]** points to name of the file that you want **display** to read. This, too, is set by COHERENT, and works automatically.

If you look below at the line that declares **fopen()**, you will see that **tester** has been replaced with **argv[1]**; this means that you want **fopen()** to open the file named in the first argument to the **display** command.

Now, try running the program by typing

```
display display.c
```

display will open **display.c** and print its contents on the screen. You still need to type **<ctrl-C>** when printing is finished; the code to recognize the end of the file will be inserted later.

Also, be sure that you give the command only one file name as an argument, no more and no less. Code that checks against errors has not yet been inserted, and handing it the wrong number of arguments could cause problems for you.

Error Checking

Obviously, the program runs at this stage, but is still fragile, and could cause problems. The next step is to stabilize the program by writing code to check for errors. To do so, a programmer must first write code to capture error conditions, and then write a routine to react appropriately to an error.

Our edited program now appears as follows:

```
/*
 * Truncated version of the 'scat' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <return>, print another 23 lines.
 * If user types any other key, exit.
 * Exit when EOF is read.
 */

#include <stdio.h>
#define MAXCHAR 128

main(argc, argv)
/* define arguments to main() */
int argc;
char *argv[];
{
        char string[MAXCHAR];
        FILE *fileptr;

/* Check if right number of arguments was passed */
        if (argc != 2)
                error("Usage: display filename");

/* Open file */
        if ((fileptr = fopen(argv[1], "r")) == NULL)
                error("Cannot open file");
```

```
/* Read material and display it */
        for (;;) {
                fgets(string, MAXCHAR, fileptr);
                printf("%s", string);
        }
}

/* Process error messages */
error(message)
char *message;
{
        printf("%s", message);
        exit(1);
}
```

The additions to the program are introduced by comments.

The first addition

```
        if (argc != 2)
                error("Usage:  display filename");
```

checks to see if the correct number of arguments was passed on the command line; that is to say, it checks to make sure that you named a file when you typed the **display** command.

As noted above, **argc** is the number of arguments on the command line, or rather, the number of arguments plus one, because the command name itself is always considered to be an argument. The statement **if(argc != 2)** checks this. The **if** statement is built into C. If the condition defined between its parentheses is true, then do something, but if it is not true, do nothing at all. The operator **!=** means "does not equal". Therefore, our statement means that if **argc** is not equal to two (in other words, if there are not two and only two arguments to the **display** command — the command name itself plus a file name), execute the function **error**. **error** is defined below.

Our **fopen** function also has some error checking added (which will be described in a moment):

```
        if ((fileptr = fopen(argv[1], "r")) == NULL)
                error("Cannot open file");
```

fopen returns a value called "NULL" if, for any reason, it cannot open the file you requested. Thus, our new **if** statement says that if **fopen** cannot open the file named on the command line (that is, **argv[1]**), it should invoke the **error** function.

C always executes nested functions from the "inside out". That means that the innermost function (that is, the function that is enclosed most deeply within the pairs of parentheses) is executed first. Its result, or what it *returns*, is then passed to next outermost function as an argument; that function is then executed and what it returns is, in turn, passed to the function that encloses it, and so on. In this instance, the innermost function is

```
        fileptr = fopen(argv[1], "r")
```

fopen is executed and what it returns is written into **fileptr**. What **fopen** returned is then passed to the next outer operation; in this case, it is compared with NULL, as follows:

```
        (fileptr = fopen(argv[1], "r")) == NULL)
```

What that operation returns is then passed to the outermost function, in this case the **if** statement, which evaluates what it is passed, and acts accordingly. If **fileptr** is NULL (that

is, if **fopen** couldn't open the file), the **if** statement will be true and the **error** function called. If, however, the file was opened, **fileptr** will not equal NULL and the program will proceed.

As this example shows, C allows a programmer to nest functions quite deeply. Although nested functions are sometimes difficult to untangle when you read them, they make programming much more convenient.

Finally, at the bottom of the file is a new function, called **error**:

```
error(message)
char *message;
{
        printf("%s", message);
        exit(1);
}
```

This function stands outside of **main**, as you can tell because it appears outside of **main**'s closing brace. This function is called only when your program needs it. If there are no errors, the program progresses only until the closing brace in **main** and the **error** function is never called.

error takes one argument, the message that is to be printed on the screen. This message is defined by the routine that calls **error**. **error** uses the function **printf** to print the message, then calls the **exit** function; this, as its name implies, causes the program to stop. The argument **1** is a special signal that tells COHERENT that something went wrong with your program.

When the error checking code is inserted, recompile the program without an argument. Previously, this would cause the program to crash; now, all it does is print the message

```
Usage: display filename
```

and terminate the program.

Print a Portion of a File

So far, our utility just opens a file and streams its contents over the screen. Now, you must insert code to print a 23-line portion of the file. At present, it will only print the first 23 lines, and then exit.

To do so, you must insert another **for** loop. Unlike our first loop, which ran forever, this one will cycle only 23 times, and then stop. Our updated program appears as follows:

```
/*
 * Truncated version of the 'scat' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <return>, print another 23 lines.
 * If user types any other key, exit.
 * Exit when EOF is read.
 */

#include <stdio.h>
#define MAXCHAR 128

main(argc, argv)
int argc;
char *argv[];
```

```
{
        char string[MAXCHAR];
        FILE *fileptr;
        int ctr;

/* Check if right number of arguments was passed */
        if (argc != 2)
                error("Usage: display filename");

/* Open file */
        if ((fileptr = fopen(argv[1], "r")) == NULL)
                error("Cannot open file");

/* Output 23 lines */
        for (;;) {
                for (ctr = 0; ctr < 23; ctr++) {
                        fgets(string, MAXCHAR, fileptr);
                        printf("%s", string);
                }
                exit(0);
        }
}

/* Process error messages */
error(message)
char *message;
{
        printf("%s", message);
        exit(1);
}
```

The new **for** loop is nested inside the loop governed by **for(;;)**. The program also declares a new variable, **ctr**, at the beginning of the program. **ctr** keeps track of how many times the loop has executed. Now, look at the line:

```
for (ctr = 0; ctr < 23; ctr++)
```

It has three sub-statements, which are separated by semicolons. The first sub-statement sets **ctr** to zero; the second says that execution is to continue as long as **ctr** is less than 23; and the third says that **ctr** is to be increased by one every time the loop executes (this is indicated by the **++** appended to **ctr**). With each iteration of this loop, **fgets** reads a line from the file named on the **display** command line, and **printf** prints it on the screen.

Also, an **exit** call has been set after this new loop. This ensures that the program will exit automatically after the loop has finished executing. This is a temporary measure, to make sure that you no longer have to type **<ctrl-C>** to return to the shell.

When you have updated the program, recompile it in the usual way. When you run it, **display** will show the first 23 lines of the file, and then the shell's prompt will return.

The program is now approaching its final form.

Checking for the End of File

The next-to-last step in preparing the program is teaching it to recognize the end of a file when it sees it. This does not appear to be needed now because the program exits automatically after 23 lines or fewer, but it will be quite necessary when the program begins to display more than one 23-line portion of text.

The function **fgets** checks to see if it has arrived at the end of a file, and returns a special value if it has. **fgets** normally returns the address of the string into which it writes its output; however, if it runs into the end of a file (or if any other error occurs), it returns the special value NULL. By reading the value of what **fgets** returns, **display** can detect if the end of the file has been encountered, and stop reading. To do so, the **fgets** statement must be set within an **if** statement. The **if** statement will capture what **fgets** returns, and continue execution as long as the value of the number returned is not NULL.

The updated program now appears as follows:

```
/*
 * Truncated version of the 'scat' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <return>, print another 23 lines.
 * If user types any other key, exit.
 * Exit when EOF is read.
 */

#include <stdio.h>
#define MAXCHAR 128

main(argc, argv)
int argc;
char *argv[];
{
        char string[MAXCHAR];
        FILE *fileptr;
        int ctr;

/* Check if right number of arguments was passed */
        if (argc != 2)
                error("Usage: display filename");

/* Open file */
        if ((fileptr = fopen(argv[1], "r")) == NULL)
                error("Cannot open file");

/* Output 23 lines, while checking for EOF */
        for (;;) {
                for (ctr = 0; ctr < 23; ctr++) {
                        if (fgets(string, MAXCHAR, fileptr) != NULL)
                                printf("%s", string);
                        else
                                exit(0);
                }
                exit(0);
        }
```

```
}

/* Process error messages */
error(message)
char *message;
{
        printf("%s", message);
        exit(1);
}
```

First, note that the comment that describes the program's output has been changed to reflect our changes to the program. It is important for a programmer to ensure that the comments and the code are in step with each other.

Our new **if** statement

```
        if (fgets(string, MAXCHAR, fileptr) != NULL)
```

checks what **fgets** returns: if it does not return NULL, the end of the file has not been reached, the **if** statement is true and the program prints out the next line. (The operator **!=** indicates "not equal".) If it returns NULL, however, the end of file has been reached, the **if** statement is false so the **else** statement is executed, which causes **display** to exit.

Note, too, that a new control statement is introduced: **else**. This, like **if**, is built into the C language. An **else** statement is always paired with an **if** statement; together, they mean that if the condition for which **if** is testing is true, the program should do one thing; otherwise, it should do something else. In this case, the program says that if the end of file has not been reached, another line should be read from the file and printed on the screen; however, if it has been reached, then the program should exit. As you can imagine, **if/else** pairs are common in C programming; they are logical and useful.

One more task must be done on our program; then it is finished.

Polling the Keyboard

For the program to be complete, it has to ask you if you want to see another 23-line portion of text. The program should write another portion if you press the **<return>** key alone; if you type any other key before you press **<return>**, then it should exit.

To do so, we will print a query on the screen, then read what the user has typed and interpret it. When these changes are inserted, the program is complete:

```
/*
 * Truncated version of the 'scat' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <return>, print another 23 lines.
 * If user types any other key, exit.
 * Exit when EOF is read.
 */

#include <stdio.h>
#define MAXCHAR 128

main(argc, argv)
int argc;
char *argv[];
{
```

```
        char string[MAXCHAR];
        FILE *fileptr;
        int ctr;

/* Check if right number of arguments was passed */
        if (argc != 2)
                error("Usage: display filename");

/* Open file */
        if ((fileptr = fopen(argv[1], "r")) == NULL)
                error("Cannot open file");

/* Output 23 lines, while checking for EOF */
        for (;;) {
                for (ctr = 0; ctr < 23; ctr++) {
                        if (fgets(string, MAXCHAR, fileptr) != NULL)
                                printf("%s", string);
                        else
                                exit(0);
                }
/* Query if user wishes to continue */
                printf("Continue? ");
                fflush(stdout);
                fgets(string, MAXCHAR, stdin);

                if (string[0] != '\n')
                        exit(0);
        }
}

/* Process error messages */
error(message)
char *message;
{
        printf("%s", message);
        exit(1);
}
```

These new lines introduce a few new twists. The lines

```
        printf("Continue? ");
        fflush(stdout);
```

print the prompt **Continue?** on the screen. Note that no '\n' appears after the the prompt; this ensures that the cursor does *not* jump to the next line, but stays next to the prompt. Because no '\n' appears after the line, however, you have to force it to appears on the screen; this is accomplished with the statement:

```
        fflush(stdout);
```

fflush flushes matter to an output device. **stdout** points to a file stream, just like the stream that you opened with the call to **fopen**, earlier in the program. **stdout** is opened in the header file **stdio.h**, which was read at the beginning of the program; it always points to the user's screen.

The next line reads the user's keyboard:

```
fgets(string, MAXCHAR, stdin);
```

This version of **fgets** reads matter into our array **string**; however, instead of reading the file pointed to by **fileptr**, it reads what is pointed to by **stdin**. **stdin** is a stream that is also defined in **stdio.h**; it always points to the user's keyboard.

Finally, the statement

```
if (string[0] != '\n')
```

checks what the user typed by reading the first (that is, the zero-th) character written in the array **string** by the preceding call to **fgets**. (Note that with C, counting always begins with zero rather than one.) If the user just types **<return>**, then **string[0]** will hold '\n'; and the **if** statement will *not* be true, the program jumps to the preceding **for** statement, and more text is written to the screen. However, if the user types anything before typing **<return>**, the **if** statement will succeed and the program will exit. This may seem a little convoluted, but it actually is a straightforward and efficient way to receive information from the user.

After you have inserted these changes, again compile the program.

When compilation is finished, try typing

```
display display.c
```

The first 23 lines of the source code to the program now appear on your screen. Hit **<return>**; the next 23 lines appear. Now, type any other key, and then press **<return>**: the program exits.

You now have a simple but helpful **display** utility.

For More Information

This section has given you a brief, concentrated introduction to writing a C program. If you are new to programming, much of what happened must seemed strange, but we hope it helped you to appreciate the logic of how C works.

Numerous books are on the market to teach beginners how to program in C; the following section gives a small bibliography of books on C. Also, look at the sample C programs in the Lexicon. These demonstrate how to use many of the functions available to you with COHERENT.

Bibliography

The following books may be helpful in developing your skills with C. This list also contains all books that are referenced in this manual. It is by no means exhaustive; however, it should prove helpful to both beginners and experienced programmers.

American National Standards Institute: *Draft Programming Language C (October 1986 Draft)*. Washington, D.C.: X3 Secretariat, Computer and Business Equipment Manufacturers Association, 1986.

AT&T Bell Laboratories: *The C Programmer's Handbook*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1985.

Bentley, J.: *Programming Pearls*. Reading, Mass.: Addison-Wesley Publishing Company, 1986. *Not, strictly speaking, about C — but belongs on every programmer's bookshelf.*

Brooks, F.P., Jr.: *The Mythical Man-Month: Essays on Software Engineering*. Reading, Mass.: Addison-Wesley Publishing Company, Inc., 1975. *Not about programming, but should be read by every programmer.*

Chirlin, P.M.: *Introduction to C*. Beaverton, Or.: Matrix Publishers, Inc., 1984.

Derman, B. (ed.): *Applied C*. New York: Van Nostrand Reinhold Co., Inc., 1986.

Feuer, A.R.: *The C Puzzle Book*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1982.

Gehani, G.: *Advanced C: Food for the Educated Palate*. Rockville, Md.: Computer Science Press, 1985.

Hancock, L.: Krieger, M.: *The C Primer*. New York: McGraw-Hill Book Publishers, Inc., 1982.

Harbison, S.; Steele, G.: *C: A Reference Manual*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.

Haviland, K.F., Salama, B.: *UNIX System Programming*. Reading, Mass.: Addison-Wesley Publishing Company, Inc., 1987.

Hogan, T.: *The C Programmer's Handbook*. Bowie, Md.: Brady Publishing, 1984.

Kelley, A.; Pohl, I.: *C by Dissection: The Essentials of C Programming*. Menlo Park, Ca.: The Benjamin/Cummings Publishing Company, Inc., 1987.

Kernighan, B.W.; Ritchie, D.M.: *The C Programming Language*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1978.

Kernighan, B.W.; Plauger, P.J.: *The Elements of Programming Style*, ed. 2. New York: McGraw-Hill Book Co., 1978.

Kochan, S.G.: *Programming in C*. Hasbrouck Heights, N.J.: Hayden Book Co., Inc., 1983.

Knuth, D.E.: *The Art of Computer Programming*, vol. 1: *Basic Algorithms*. Reading, Ma.: Addison-Wesley Publishing Co., 1969.

Knuth, D.E.: *The Art of Computer Programming*, vol. 2: *Seminumerical Algorithms*. Reading, Ma.: Addison-Wesley Publishing Co., 1969.

Knuth, D.E.: *The Art of Computer Programming*, vol. 3: *Sorting and Searching*. Reading, Ma.: Addison-Wesley Publishing Co., 1969.

Lapin, J.E.: *Portable C and UNIX System Programming*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1987.

Mark Williams Company: *ANSI C: A Lexical Guide*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

Plum, T.: *C Programming Guidelines*. Cardiff, N.J.: Plum Hall, Inc., 1984.

Plum, T.; Brodie, J.: *Efficient C*. Cardiff, NJ: Plum Hall, Inc., 1985.

Purdum, J.: *C Programming Guide*. Indianapolis: Que Corp., 1983.

Purdum, J.; Leslie, T.C.; Stegemoller, A.L.: *C Programmer's Library*. Indianapolis: Que Corp., 1984.

Rochkind, M.J.: *Advanced UNIX Programming*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1985.

Traister, R.J.: *Going from BASIC to C*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1984.

Traister, R.J.: *Mastering C Pointers*. New York: Academic Press, Inc., 1990.

Traister, R.J.: *Programming in C for the Microprocessor User*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1984.

Vile, R.C., Jr.: *Programming in C with Let's C*. Glenview, IL: Scott, Foresman and Company, 1988.

Waite, M.; Prata, S.; Martin, D.: *C Primer Plus*. Indianapolis: Howard W. Sams, Inc., 1984.

Weber Systems, Inc.: *C Language User's Handbook*. New York: Ballantine Books, 1984.

Zahn, C.T.: *C Notes*. New York: Yourdan Press, 1979.

Section 6:

Introduction to ed, Interactive Line Editor

This tutorial introduces the interactive editor **ed**. It is intended both for readers who want a tutorial introduction to **ed**, and those who want to use specific sections as a reference.

Related tutorials include include those for **sed**, the stream editor, and for **me**, the MicroEMACS screen editor. This tutorial assumes that you already understand the basics of using the COHERENT system, such as what a file is, what it means to edit text, and how to issue commands to the operating system. If you not yet know your way around the COHERENT system, we suggest that you first study the *Using the COHERENT System*, which appears in the front of this manual. It covers the basics of using COHERENT and introduces many useful programs.

Why You Need an Editor

A significant feature of computers is the capacity to store, retrieve, and operate upon information. A computer can store many different kinds of information: programs, computer commands and instructions, data for programs, financial information, electronic mail, or natural-language text (e.g., French, English) destined for a manuscript or book.

ed is a program with which you can enter and edit text on your computer. You can use **ed** to create or change computer programs, natural-language manuscripts, files of commands, or any other file that consists of text that you can read.

ed is designed to be easy to use, and requires little training to get started. The fundamental commands are simple, but have enough flexibility to perform complex tasks.

Learning To Use the Editor

Practice on your part will help you learn quickly. The following sections contain examples that illustrate each topic discussed. We strongly recommend that you type each example presented as you encounter it in the text. Even if you understand the concept presented, performing the example reinforces the lesson, and you will learn more quickly how to use **ed**.

In addition to reading the text and doing the examples as you encounter them in the text, try your own variations on the commands, and branch out on your own. Try things that you suspect might work, but are not shown as examples.

General Topics

This section presents the backround information you will need to understand how **ed** works.

To help illustrate the discussion to follow, log into your COHERENT system and type the following commands:

```
ed
a
this is a sample
ed session
.
w test
q
```

This example calls **ed**, then uses the **a** command to add lines to the text kept in memory. The period signals the end of the additions. The **w** command writes the lines of text to file **test**, and the command **q** tells **ed** to return to COHERENT. You will notice that after you type the **w** command, **ed** will respond with

```
28
```

which is the number of characters in the file.

Thus, to enter **ed**, simply type

```
ed
```

and to exit, type

```
q
```

You can also exit by typing **<ctrl-D>**: that is, hold down the **control** key on your keyboard, and at the same time strike the **D** key.

Notice that you are issuing two different kinds of commands in the above example. The command **ed** is an COHERENT command, whereas the rest are commands to the editor. After **ed** is given the **q** command, it exits, and following commands are processed by COHERENT.

ed, Files, and Text

ed works with one file at a time. With **ed**, you can create a file, add to a file, or change a file previously created.

As you use **ed** to create or change files, you will type both *text* and controlling *commands* into the editor. Text is, of course, the matter that you are creating or changing. Commands, on the other hand, tell **ed** what you want it to do. As you will see shortly, there is a simple way to tell **ed** whether what you are typing is text or commands.

ed has about two dozen commands. Almost every one is only one letter long. Although they may seem terse, they are easy to learn. You will appreciate the brevity of the commands once you begin to use **ed** regularly.

You must end each command to **ed** by striking the **<return>** key. This key is present on all terminals. However, the labeling of the key may vary. It may be called **newline**, **linefeed**, **enter**, or **eol**, and is larger than any key on the keyboard except for the space bar. This key will be called the **<return>** key in the remainder of this document.

Creating a File

The example shown above created a file. Here is another example of file creation here, creating a file called **twoline**:

```
ed
a
Two line Example,
thank you.
.
w twoline
q
```

The letter **a** tells **ed** to add lines to the file. You are creating a new file with this example; and when **ed** creates a new file, it is initially empty. The **w** command writes the lines you have added to file **twoline**. The command **q** tells the editor that you are finished, whereupon it returns to COHERENT. You can use the COHERENT command **cat** to list the contents of the new file:

```
cat twoline
```

the reply will be:

```
Two line Example,
thank you.
```

Each command used here will be described in detail in later sections.

Changing an Existing File

Suppose that a manuscript file of yours needs a few spelling corrections. **ed** will help you make them. To begin, simply name the file to correct when you issue the COHERENT command:

ed *filename*

where *filename* stands for the name of the file that you wish to edit. For example, the following adds a line to the file *twoline*, which we just created:

```
ed twoline
$a
This is the third line of the file.
.
w
q
```

Listing the file with **cat** gives:

```
Two line Example,
thank you.
This is the third line of the file.
```

The command **$a** tells **ed** to add one or more lines at the end of the file.

Correcting the spelling of a misspelled word is easy with **ed**. You can rearrange groups of words in a manuscript, and you can move or copy larger portions of text, such as a paragraph, from one spot to another.

Working on Lines

ed uses the *line* as the basic unit of information; for this reason, it is called a *line-oriented* editor. A line is defined as a group of characters followed by an end-of-line character, which is invisible. When you type out a file on your terminal, each line in the file will be shown on your terminal as one line. The commands for **ed** are based upon lines. When you add material to a file, you will be adding lines. If you remove or change items, you will do so to groups of lines.

ed knows each line by its number. A line's number, in turn, indicates its position within the file: the first line is number 1, the second line is number 2, and so on.

ed remembers the line you worked on most recently. This can help shorten the commands you type, as well as reduce the need for you to remember line numbers. The line most recently worked on is called the *current* line. **ed** commands use a shorthand symbol for the current line: the period '.'.

Another shorthand symbol used in **ed** commands is **$**, which represents the number of the last line in the file.

Many of the **ed** commands operate on more than one line at a time. Groups of lines are denoted by a range of line numbers, which appears as a prefix to the command.

Error Messages

If you type a command to **ed** incorrectly, **ed** respond with:

```
?
```

This indicates that it has detect an error. Many times, this error will be evident to you when you review the command that you just typed.

If you do not see what the error is, you can get a more lengthy description by typing to **ed**:

```
?
```

It will reply with an error message.

Basic Editing Techniques

This section discusses in more detail the elementary techniques and commands that you need to use **ed**. With the material presented in this section, you will be able to do most basic editing tasks.

Again, it is recommended that you type each example. This will help you understand each example, as well as remember the technique it demonstrates.

Creating a New File

To begin, let us presume that you need to create an entirely new file named **first**. Perhaps you only want one line in the file, and it is to read

```
This is my first example
```

These are the steps that you will need to go through to create this file.

The first step is to invoke the **ed** program. To do this, simply type

```
ed
```

Remember that you must end each line of commands or text line by pressing the **<return>** key, because **ed** will not act upon it until you do. Thus, you invoke the editor by typing **ed** and a **<return>**. Notice that these two characters must be lower case.

ed is now ready for commands. The first command that you will use is the append command **a**. This tells **ed** to add lines to the text in memory, which will later be written to the file. The number of lines that **ed** can hold in memory depends upon the amount of memory in your computer. For editing very large files, you should use **sed**, the COHERENT stream editor, which is described in in its own tutorial.

ed will continue to add lines until you type a line that contains *only* a period. While it is adding lines, **ed** does not recognize commands.

After you issue the **a** command, you can type the lines to be included, concluding with a line that consists only of a period. This special line signals **ed** that you want to stop appending lines. The information that you have typed so far is:

```
ed
a
This is my first example
.
```

Next, you must tell **ed** to write the edited text into a file. Do so by issuing the write command **w**, plus the name of the file that is to hold the edited text. For example, if you wish to store this example in a file named **first**, issue the command:

```
w first
```

ed will write the file and tell you how many characters were written, in this case 25.

Finally, to quit the editor issue the quit command:

```
q
```

The commands you type after this will be interpreted and acted upon by COHERENT.

Now, review the example in its entirety. First you invoked **ed** by typing **ed** at the COHERENT prompt. Then you issued the add command **a** to add lines to the file. added lines with the **a** command, and finished the adding by typing a line that consists only of a period. You then wrote the editing text into a file by issuing the write command **w**, and finally you exited from **ed** by issuing the quit command **q**. The complete example is:

```
ed
a
This is my first example
.
w first
q
```

ed replied to the **w** command by printing the number of characters it wrote into the file. After you typed **q**, COHERENT prompted you for a command again.

Changing a File

Suppose that you wish to change the file that you have just created: you want to add two more lines to the file so that the original line will be sandwiched between the new lines. You want the file to contain:

```
Example two, added last
This is my first example
Example two, added first
```

You will do this with **ed** using two new commands.

Again, you start by telling COHERENT to run **ed**. This time, however, you must type the name of the file that you are changing after the characters **ed**:

```
ed first
```

ed will remember this file name for later use with the **w** command.

ed reads the file in preparation for editing, and tells you the number of characters that it read in, again 25.

After reading the file, **ed** automatically sets the current line to the last line read in.

Now, add the third line shown in the second example by entering:

```
a
Example two, added first
.
```

This resembles the first example. In that case, however, the file had no information, whereas now it does. How did **ed** know where to add the lines?

The **a** command adds lines after the *current line*. When **ed** reads a file, it initially sets the current line to the last line read in; therefore, the **a** command added the new line after the last line.

The current line is used implicitly or explicitly by most commands, so it is helpful to know where it is. In general, the current line is left at the last line **ed** has processed. If you lose track of the current line, you can ask **ed** to tell you where it is, as you will see shortly.

To add the very first line to the second example, you will use yet another command, the insert command **i**. This command is identical to the **a** command, except that it inserts lines *before* the current line rather than after it.

Another word about the current line. After an **a** command finishes, the current line is the last line added. Thus, after the addition of "Example two, added first" above, the current line is now the last line in the file. So, if you were to do the **i** command immediately, you would be adding lines just before the last line, which is not what you want to do.

Nearly every **ed** command is flexible enough to allow you to specify the line upon which the command is to operate. Now you can complete the second example:

```
1i
Example two, added last
.
```

The numeral **1** before the **i** tells **ed** to insert lines before the first line in the file. The line-number prefix is used frequently, and applies to nearly every command.

Now, to finish the second example and save it into the same file, type:

```
w
q
```

Note that the file name was left off the **w** command. **ed** remembers the name of the file that you began with, and uses that name if none is used with the **w** command. Therefore, the edited text is written back into file **first**. Note, too, that the previous contents of the file **first** are lost when you write the new file **first**. Alternatively, you can type:

```
w second
```

This leaves the contents of **first** unchanged and creates a new file called **second**.

In case you forget, **ed** can tell you the name of the file with which you began. Simply type the command:

```
f
```

If you had used **f** any time while working on this second example, **ed** would have replied:

```
first
```

Remember to use the **q** command to leave **ed** and return to COHERENT.

Printing Lines

As you use **ed** to edit a file, you will find it most useful to print sections of the file on your terminal. This helps you see what you have done (and sometimes what you have not done), and helps you pinpoint where you wish to make changes.

The print command **p** prints the current line unless you specify a line number.

Continuing with the example begun above, when you type the commands

```
ed first
p
```

ed replies by printing

```
Example two, added first
```

which is the last line in the file named **first** from the previous example.

Again, like the commands **i** and **a**, if you want **ed** to print a line other than the current one, just prefix the **p** command with a line number. Thus, if you want to print the second line in the file, type:

```
2p
```

ed will reply with:

```
This is my first example
```

If you wish to print more than one line of a file, you can tell **ed** to print a *range* of line numbers: type the numbers of the first and last lines you wish to see, separated by a comma. For example, to print all three lines in the second example, type:

```
1,3p
```

ed responds by printing all lines. This same principle applies to other commands. The print command can also appear after other commands such as **s** or **d**, which are discussed

later in this section.

Abbreviating Line Numbers

ed recognizes some shorthand descriptions for certain line numbers. The number of the last line can be represented by the dollar sign **$**. Thus, the command

```
1,$p
```

prints every line in the file. The advantage of this shorthand is that the command as typed works for any file, regardless of its size. This construct of **1,$p** is used often enough that it has an abbreviation of its own:

```
*p
```

The number of the current line can also be abbreviated by using the period or dot in the place of a line number. To print all lines from the beginning of the file through the current line, type:

```
1,.p
```

To print all lines from the current line through the end of the file, type:

```
.,$p
```

The special symbol **&** prints one screenful of text. Simply type:

```
&
```

This is equivalent to:

```
.,.+22p
```

If there are fewer than 23 lines between the current line and the end of the file, it is equivalent to

```
.,$p
```

All forms of the **p** command change the current line to the last line printed. The command

```
.,$p
```

after printing changes the current line to the last line of the file.

How Many Lines?

You can easily see the current line with **p**. Type:

```
p
```

This tells **ed** to print the current line. On your terminal, try the command:

```
.p
```

You will see that it does the same thing as **p**.

To discover how large your file is, just type:

```
=
```

ed will reply by typing the number of lines in the file.

To find the number of the current line, use the **dot equals** command:

```
.=
```

ed responds with the number of the current line.

Removing Lines

Editing means removing lines of text, as well as adding them. To illustrate how **ed** lets you remove lines of text, create another example file with **ed**:

```
ed
a
This is the first line.
The second line is good.
However, line three is bad.
line four wishes to go away.
line 5 similarly wants to be forgotten,
as does line 6.
the next to last line stays.
as does the last line in the file.
.
w example3
q
```

This creates a file named **example3**.

Now, you can practice removing lines that you no longer want. Begin editing the file by typing:

```
ed example3
```

Now, print the contents of the file by typing:

```
1,$p
```

Our first task is to delete lines 3 through 6. First, delete line 3, then print the entire file again.

```
3d
1,$p
```

and **ed** will respond with

```
This is the first line.
The second line is good.
line four wishes to go away.
line 5 similarly wants to be forgotten,
as does line 6.
the next to last line stays.
as does the last line in the file.
```

Notice that the original file's third line is no longer there. Line 3 is now what used to be line 4. Remember that the line numbers *always* begin with 1 for the first line of the file and progress consecutively even after the file has been changed. Thus, deleting a line will change the line number of each line from the deleted line to the the last line in the file.

You still need to remove three more lines. You can do this with one command:

```
3,5d
```

Again, type ***p** to print the contents of the file:

```
This is the first line.
The second line is good.
the next to last line stays.
as does the last line in the file.
```

Finally, write the updated file and quit:

```
w
q
```

This illustrates how to delete lines, both singly and in a group.

Abandoning Changes

Sometimes, you may made a mistake; rather than damage your file with badly edited text, you may wish to abandon what you have done and begin all over again. You can do so by using the **q** command in a different fashion than is shown above.

If you tell **ed** to **q** before you tell it to write the file with **w**, you abandon any changes made since beginning editing. However, to prevent you from accidentally selecting this option, **ed** checks to see if you have made any changes to the file; and if you have, it responds with a question mark '?'. To tell **ed** that you know what you are doing and really do wish to abandon the edited file, reply with a second **q**. **ed** will then quit and return you to COHERENT.

You can avoid the question mark prompt by typing the upper-case **Q** rather than lower-case **q**: **ed** will exit without regard to unsaved changes. You can also exit from **ed** by typing the end-of-file key **<ctrl-D>**.

Substituting Text Within a Line

If you type a line incorrectly, or later wish to rearrange some words or symbols within it, you know enough about **ed** now to do so. You only need to delete the line with the delete command **d** and re-type the line with the insert command **i**. To see how this is done, prepare the file **example4**, as follows:

```
ed
a
Software technology today has
adbanced to the point that large
software projects unherd of in
earlier times are undertaken and
.
w example4
q
```

This example has two misspelled words. We will correct each of them using different **ed** features.

The first method will be the direct way that you probably can anticipate. Give the following commands to the editor exactly as shown:

```
ed example4
2d
i
advanced to the point that large
.
```

These commands use the delete command **d** to delete the second line, and then uses the insert command **i** to insert the correct new line in its place.

Use the command

```
*p
```

to verify that the file now contains:

```
Software technology today has
advanced to the point that large
software projects unherd of in
earlier times are undertaken and
```

You can also use a second method to change the spelling of a word. This is the substitute command **s**. This command is very powerful, and probably is used more frequently than any other **ed** command.

The substitute command **s** is more complex than commands we have discussed so far, in that it has more elements, as follows: First is a line number or optional range of line numbers. Then comes the letter **s**, to invoke the substitute command itself. Third comes two *patterns* or *strings*, which are set off from the rest of the command and from each other with the **slash** character. For example:

```
1,$s/pattern1/pattern2/
```

Here, *pattern1* represents the string that you want **ed** to replace, and *pattern2* is the string that **ed** is to substitute in place of *pattern1*. Note that three slashes separate the two patterns from the **s**, from each other, and from the end of the line. These slashes must always be present.

With this command, you can correct the second spelling error in the example4:

```
3s/herd/heard/
p
```

ed replies:

```
software projects unheard of in
```

Note that these two command lines can be condensed to one:

```
3s/herd/heard/p
```

The meaning of these commands is: on the third line of the file, change **herd** to **heard** and, when finished, print the entire line. Without the **p** command, **ed** will change the line as you direct, but will not show you the new line. It is a good idea to print lines that you substitute in this manner until you gain in confidence with **ed**. Some **ed** experts always print the lines after substitution.

After these two changes, the file will look like this:

```
Software technology today has
advanced to the point that large
software projects unheard of in
earlier times are undertaken and
```

Although the above example substitutes one word for another, note that the **s** command can replace any conseutive group of characters with any other: it may be one word, several words (including the space characters that separate them), or a fragment of a word.

Because **ed** looks for patterns rather words, you should keep in mind that it may find the wrong pattern. For example, assume that the current line in a file is

```
let not rain fall on a parade
```

and instead you want to say:

```
let not rain fall on the parade
```

You command **ed** to:

```
s/a/the/p
```

and are shocked to discover that the result is:

```
let not rthein fall on a parade
```

A better command to give **ed** would have been a substitute command that substituted the letter **a** preceded and followed by a space:

```
s/ a / the /p
```

Another correct way to do this task is to indicate within the substitution command which of several possible matches within the line is to be substituted. In our example, it is actually the third **a** that we are trying to match, so we could have used the special form of the command

```
s3/a/the/p
```

to get **ed** to select the one we wanted.

Undoing Substitutions

If you did change **a** to **the** inappropriately, you can retract the substitution by issuing the undo command

```
u
```

before you move on to another current line.

To illustrate this, enter this example:

```
ed
a
let not rain fall on a parade
.
w undo
q
```

Now, perform the substitution with

```
ed undo
s/a/the/p
```

which will result in:

```
let not rthein fall on a parade
```

To retract the substitution, simply type:

```
u
p
```

This undoes the substitution and prints the result.

Note that the undo command undoes the substitution only on the current line. Remember that if your substitution command operated over a range of lines, when it finishes the current line is the last one upon which the substitution was made. Thus, if you made an inappropriate substitution over a range of lines, the undo command will fix only the last line.

Global Substitutions

As you saw with the above examples, the **s** command substitutes only the *first* occurrence of the requested pattern on a given line.

A different form of the substitute command finds every occurrence of the indicated string on a line. Simply add the letter **g** for *global* after the third slash in the substitute command, and **ed** finds and changes every one:

```
s/pattern1/pattern2/g
```

So, if the current line contains a phrase:

```
a rose is a rose is a rose
```

and we tell **ed** to substitute

```
s/a/the/g
```

the line is changed to:

```
the rose is the rose is the rose
```

Again, be careful that your command does not inadvertently match all or part of a word that you wish to keep untouched.

Special Characters

In its first two parts, the substitute command uses some special punctuation characters. They will be discussed below in detail. However, you should be aware of these characters and avoid them until you progress to the advanced section, for unless used properly, they will give you undesired results. The characters are:

```
[ ^ $ * . \ &
```

They are used in **ed** and other COHERENT programs to form complex patterns.

Ranges of Substitution

Perhaps you need to change several lines that have the same misspelling or need the same editorial change. **s** can do that for you also. Simply prefix the command **s** with the line-number range as you would do with **p**. Borrowing the "rose" example again, if the saying were typed:

```
a rose is
a rose is
a rose
```

then you could do the same change as before, but across the entire file by typing

```
1,$s/a/the/
```

Note that the **g** after the **s** command has been omitted here, because you know that the string that you want to change appears only once on each line.

If some of the lines do not have the the string you want to change, **ed** will not complain that the string is missing. However, if none of the lines in the range has the requested string, **ed** will print a **?**.

Intermediate Editing

This section introduces the more advanced command features of **ed**. Although you have already learned enough about **ed** to become productive, this section covers additional features that will increase your editing power considerably.

This section discusses the following topics: relative line numbering, moving blocks of text, finding strings, using special characters in substitution and search commands, processing global commands, and marking lines.

Relative Line Numbering

As discussed in the previous section, most commands allow you to use line numbers to control their range of operation. Before the command you can enter a single line number; for example:

```
1p
```

This, of course, prints the first line of the file. You may also specify a *range* of line numbers, by entering two numbers separated by a comma. For example, if the file contains at least ten lines, the command

```
1,10p
```

prints the first ten lines of the file.

The period (dot) always represents the number of the current line. For example, to print the file from the first line through the current line, just type:

```
1,.p
```

A command used without a line number always acts on the current line only. For example, typing

```
p
```

is equivalent to typing:

```
.p
```

There is yet another level of shorthand to line numbering the plus and minus characters. These characters indicate *offsets* from the current line. For example, the command

```
.+3p
```

prints the third line after the current line. Likewise, the command

```
.-1p
```

prints the line that precedes the current line. Note that using a line offset changes the current line to the one addressed. Thus, after the above command is executed, the current line will be the one that preceded the original current line.

You can abbreviate this notation still further by leaving out the dot. The commands

```
+p
-p
```

do the following: First, **ed** advances to the next line and prints it; then it backs up to the previous line (which was the original current line) and printing it.

You can place several of these commands on one line to move the current line multiple lines. To back up three lines and then print, type:

```
---p
```

Note that in the absence of any other command, **ed** defaults to the **p** command. Thus

```
---
```

is equivalent to

```
---p
```

and

```
5
```

is identical to:

```
5p
```

The print command has one more abbreviation. If **ed** is expecting a command from you and you type nothing except **<return>**, **ed** interprets this as a command to advance the current line to the next line and print it. This action is equivalent to

```
+
```

or

```
.+1
```

<return> is the shortest command in **ed**.

All of the abbreviations for line numbers can be used by other commands that expect a range of line numbers. For example, if you want to delete five lines centered about the current line, you could type:

```
.-2,.+2d
```

and you would get your wish.

Note that **ed** does not allow you to specify a line number that is beyond the range of the file; this is regardless of whether you are typing a line number or any form of abbreviated line numbering. For example, suppose the current line is the last line in the file and you type:

```
+
```

This tells **ed** to "advance one line then print"; however, this is impossible because you are at the last line of the file, so there is no next line to print. When you request an impossible line number, **ed** replies by printing a question mark. Note, however, that the current line is always be valid so long as the file has at least one line in it. Thus, unless the file is empty, the command

```
.
```

will never give an error message.

Changing Lines

Earlier, an example of spelling correction was solved two ways. The first way was the clumsy way of deleting a line and retyping the entire line. This strategy means much work to change a single letter, so the substitute command was introduced instead.

On occasion, however, it is handy to be able to change lines en masse as was done by deleting then inserting. **ed** provides this power with the change command **c**. In general terms,

```
m,nc
new lines
to be inserted
.
```

removes lines *m* through *n*, and insert new lines up to the period in place of them.

Moving Blocks of Text

When handling text, you will often need to shift a block of text from one position to another. In a manuscript, for example, you may need to rearrange the order of paragraphs to increase clarity . In a program, you may need to rearrange the order in which procedures appear.

To allow you to do this easily, **ed** provides a move command **m** that moves a block of text from one point in the file to another.

m is different from the other commands that we have discussed so far, in that line numbers follow as well as precede the **m** command itself. The line number that follows the command gives the line *after* which the text is to be moved. So, the general form of the move command is

```
b,emd
```

which means "move lines *b* through *e* to after line *d*".

To see how this works, first build the following file:

```
ed
a
    This is a paragraph of natural language
text. Due to stylistic considerations, it
really should be the second paragraph.
    If you can read this paragraph first,
the text has been properly arranged, and
our move example has been successfully done.
.
w example5
q
```

The file **example5** contains two paragraphs, each three lines long. We will now move the first paragraph to after the second paragraph.

You can do this in either of two ways: you can move the first paragraph to after the second paragraph, or you can move the second paragraph to before the first paragraph. Either gives the same result, but the commands are somewhat different. To shift the first paragraph to after the second paragraph, type:

```
ed example5
1,3m$
*p
Q
```

Remember that **$** always represents the last line in the file. The result is:

```
    If you can read this paragraph first,
the text has been properly arranged, and
our move example has been successfully done.
    This is a paragraph of natural language
text. Due to stylistic considerations, it
really should be the second paragraph.
```

To move the second paragraph to before the first, type:

```
4,6m0
```

Note that the destination is 0, which means that the text is to be moved to immediately after line 0. Because there is no line number 0, the move command interprets this to mean the beginning of the file.

Of course, in our small example, line number abbreviations and knowledge of the current line may be used in a number of different ways to perform exactly the same action. For example,

```
1,3m.
```

says to move lines 1 through 3 of the file to the line after the current line. When you invoke **ed**, it always sets the line number to the last line in the file. Thus, this form of the command has the same effect as the previous forms.

If the destination of a move command is not specified, **ed** assumes the current line. Therefore, the command

```
1,3m
```

also repositions the first paragraph correctly.

The move command changes the line numbers in the file, although the number of lines in the file remains the same. The different forms of the move command will, however, yield different settings for the current line.

After a move command, the current line becomes the number of the last line moved. Thus, if you moved the first paragraph to after the second paragraph, the current line will be reset to the last line in the file the original line 3. However, if you moved the second paragraph to before the first paragraph, the current line would be reset to line 3 which was originally the last line in the file.

Copying Blocks of Text

The transfer command **t** resembles the move command, except that it copies text rather than moving it. When you move text, it is erased from its original position. When you copy text, however, the text then appears both in its original position and in the position to which you copied it. **ed** uses the term *transfer* rather than *copy* because the command **c** is already used by change command.

The form of the transfer command is as follows:

```
b,etd
```

This means to transfer (copy) the group of lines that begins with *b* and that ends with *e* (inclusive) to after line *d*.

After copying the text, **ed** sets the current line to the last line copied.

String Searches

The methods of line location that have been discussed to this point all involve line numbers. They specified an absolute line number, a relative line number, or a shorthand symbol such as **.** or **$**.

Often, however, line numbers are not useful, because there is no easy way to tell what number a line has, how many lines ago a block of text began, and so on.

ed's solution to this problem is to locate a line by asking **ed** to search for a pattern of text. **ed** begins searching on the line that follows the current line, and looks for a line that matches the specified pattern. If it finds a line that contains the requested pattern, **ed** resets the current line to that line.

If **ed** encounters the end of the file before finds a match, **ed** jumps to the first line in the file, and continues its search from there. If it finds no match by the time it returns to the line where the search began, **ed** gives up and issues an error message the question mark **?**. Remember, if you type a question mark in response to an error message, **ed** will tell you in more detail what the error is.

What does it mean to "match" a pattern? The simplest meaning is that two patterns are the same the strings have exactly the same characters in exactly the same order. To see how this works, type the following to create file **example6**:

```
ed
a
    This is an example that we will
use for string searching.  There
is much natural language here as well
as some genuine arbitrary strings.
890,;+  foxtrot
qwertyuiop ##
.
w example6
q
```

Now, to locate and print any line contains the pattern **fox**, type:

```
ed example6
/fox/p
```

In response, **ed** prints the line:

```
890,;+  foxtrot
```

Also, you can use string expressions to print a range of lines. For example:

```
ed example6
/This/,/much/p
```

This prints:

```
  This is an example that we will
use for string searching. There
is much natural language here as well
```

That is, it printed all lines from the first line that contains the pattern **This** through the first line that contains the pattern **much**.

Pattern searches can also be combined with relative line numbers. If you have a Pascal program file with several procedures in it, but you find that you need to rearrange the procedures, you can combine the power of the move command with the string searches.

```
PROCEDURE A;
...
...
PROCEDURE B;
...
...
PROCEDURE C;
```

Assume that the section of text that begins with **PROCEDURE A** should follow the line that contains **PROCEDURE B**. The following command moves the text properly:

```
/PROCEDURE A/,/PROCEDURE B/-1m/PROCEDURE C/-1
```

This commands **ed** (1) to locate the chunk of text that begins with a line containing the pattern **PROCEDURE A** and ends with the line just before the first line that contains the pattern **PROCEDURE B**, and then (2) move that text to just before the first line that contains the pattern **PROCEDURE C**. As you can see, you can pack a lot of information into one **ed** command.

Let's look at this command in more detail, to see exactly how it works. First, remember that the move command **m** is defined as

b,emd

where *b* indicates the first line of the text to be moved, *e* indicates the last line of the text to be moved, and *d* indicates the line that the moved text is to follow. Thus, *b* corresponds to the number of the line that contains **PROCEDURE A** and is the first line of the procedure in question. *e*, however, corresponds to the line before the **PROCEDURE B** begins, by virtue of the -1. Here is an example of mixing pattern searches with relative line numbers, as mentioned above. Thus, you have found the beginning and ending lines of procedure A.

The final string search locates the first line of subroutine C. The move command normally moves text to **after** the given line; and because we wish to move the text to *before* the line that contains **PROCEDURE C**, we must include the -1 to move the text up one line.

Remembered Search Arguments

As discussed earlier, line numbers may be abbreviated in many ways. They may be entered as **.**, or **+**, or **-**, and certain combinations of these. With some commands, pressing **<return>** tells **ed** to use the current line number.

ed encourages you abbreviate the search string. If you enter no string between the slashes in a search or substitution, then **ed** uses the last-used search string. A common use is in the global substitution command (which will be discussed in detail later in this section):

```
g/please remove this string/s// /p
```

This does not quite remove it, but replaces it with a blank. The last-used string can be specified by a string search, a substitute command, or a reverse string search (also discussed later in this section). Also, the remembered search argument may also be used in any one of these. You can use the remembered search feature to "walk" through the file, finding the next occurrence of a remembered search pattern.

Uses of Special Characters

As powerful as the line locator seems, some features are even even more powerful. These will be discussed in the Expert Editing section, below. However, these more powerful capabilities depend upon certain punctuation marks used in a special way. As you use the line locator (as well as the substitute command), be aware of these following characters:

```
[ ^ $ * . \ &
```

They have special significance to **ed** when they appear in a string search or a substitution pattern.

If you need to use one of these characters without invoking its special meaning, precede it with a backslash '\'. This tells **ed** not to interpret the character in a special way.

For example, to find a backslash character, type the search command:

```
/\\/
```

If any of these characters is to be used in another context, for example, within lines that you are adding with the **a** command, it should *not* be preceded with the backslash. Only use the backslash to hide the meaning when it appears within the string search command, or within the first part of the substitution command.

Global Commands

The global commands **g** and **v** let you repeat commands on all lines within a specified range. For example, to print all lines that contain the word **example**, type:

```
g/example/p
```

The global command can be prefix almost any command. For example, the following command deletes all lines that contain three consecutive plus signs:

```
g/+++/d
```

Likewise, the command

```
g/foxtrot/.-2,.+2p
```

prints the five lines that surrounds any line that contains the word **foxtrot**.

A common use of the global command is to perform global substitution. The command

```
g/PROCEDURE/s/PROCEDURE/PROC/gp
```

performs the substitution on each line that contains the string **PROCEDURE** and prints the resulting line.

This may appear similar to the command

```
1,$s/PROCEDURE/PROC/gp
```

but is different in that the global command prints each of the changed lines, whereas the substitute command prints only the last line changed. Also, the method of operation of these two commands is different.

A related command **v** performs much the same task, but executes the commands only for lines that do *not* contain the specified string. Thus, to print all the lines that do not have the letter **w**, use:

```
v/w/p
```

For more sophisticated uses of the **g** and **v** commands and how they work, see the section on Expert Editing.

Joining Lines

What do you do if you inadvertently hit **<return>** as you are adding lines and need to combine the two lines?

```
ed
a
Look out, I seem to have hit ret
urn in the
middle of a word and don't know
what to do!
w rid
q
```

Rather that retyping the entire line, you can use the join command **j**:

```
ed rid
1,2j
1,$p
```

This will gives:

```
Look out, I seem to have hit return in the
middle of a word and don't know
what to do!
```

If no line number is specified, **j** joins the current line and the following line. If a single line number is specified, join operates on that and the following line.

Several lines can be joined by using the form of the command:

a,*b*j

This joins lines *a* through *b* into one line. Likewise, the command

1,$j

joins all the lines in the file into one line. Then, the command **.p** or **p** prints the entire file.

Note that the command

3j

does the same job as the command

3,4j

The join command generates its own second line number if none is specified, so that the command

*n*j

is equivalent to

n,*n*+1j

where **n** is a line number. This command is the only one that interprets a missing line number this way.

Splitting Lines

You can split one line into two with the substitute command **s**. To illustrate, suppose you typed in the following commands:

```
ed
a
This line wants to be two, with this second.
.
w split
q
```

To perform the split, type:

```
ed split
s/two, /two,\
/p
*p
wq
```

The line split is caused by the backslash that precedes the **<return>**. This tells **ed** that the **<return>** does not terminate the command, but that it is part of the substitution. The contents of file **split** are now:

```
This line wants to be two,
with this second.
```

Marking Lines

As you edit a manuscript or program, it is sometimes handy to be able to leave a "bookmark" in the text for later reference. **ed** provides this feature with the mark command **k**. To mark the next line that has the word **find**, use

```
/find/ka
```

where the letter **a** is the mark. To print the line that has been so marked, use:

```
'ap
```

You can place these references anywhere that a line number is expected.

The mark must be one lower-case letter. Also, each mark is associated with one line. Marking a line with the **k** command does not change the current line.

Marks can be especially handy in you move paragraphs with the **m** command. They give you a chance to review the sections that you will be moving before you do the move.

For example, suppose that you have a manuscript with a paragraph that must be moved to a different part of the document. Create the following example:

```
ed
a
    This is a paragraph, first line, that
needs to be moved.
text
text
And this is the last sentence of the paragraph.
    Next paragraph begins here.
text
text
text
    This is the spot that we want the paragraph
to precede.
.
w example7
q
```

Now, place three marks to help with the move:

```
ed example7
/first line,/ka
/Next paragraph/kb
/is the spot/kc
```

This marks the first line to be moved with **a**, the line after the last to be moved with **b**, and the the paragraph's destination with **c**. But you can see that the move command moves lines to the line *after* the third number specified, so let's change the third mark:

```
'c-1kc
```

Now we can use **c** in the move command without arithmetic. Now, print the paragraph to be moved to be sure that the marks are correct.

```
'a,'bp
```

ed replies with

```
    This is a paragraph, first line, that
needs to be moved.
text
text
And this is the last sentence of the paragraph.
    Next paragraph begins here.
```

You can see that we would move one line too many if we used the marks as they are. So, change **b** also.

```
'b-1kb
```

Now, do the move:

```
'a,'bm'c
1,$p
```

The file now contains:

```
    Next paragraph begins here.
text
text
text
    This is a paragraph, first line, that
needs to be moved.
text
text
And this is the last sentence of the paragraph.
    This is the spot that we want the paragraph
to precede.
```

Marking sections of text can increase the ease with which you solve your complex **ed** problems.

Searching in Reverse Direction

All scanning, processing, and searching has been shown going from the beginning of the file toward the end. Sometimes it is useful to find some word that occurs **before** the current line.

You can get **ed** to do string searching in the reverse direction by specifying the search with question marks **?** rather than slashes /. To find the previous occurrence of the word **last**, use:

```
?last?
```

This form of searching can be useful in finding the beginning and end of a **repeat/until** statement. For example, if the current line is in the middle of a Pascal **repeat/until** group, you can print the group with the command:

```
?repeat?,/until/p
```

The reverse search is like the forward search in every way except the direction of search. The search begins one line before the current or specified line, and proceeds toward the beginning of the file. If the string is not found by the time that the search reaches the beginning of the file, the search resumes at the end of the file, and progresses towards the starting point of the search. If the string is not found when the search reaches the original starting point, the question-mark error message is issued signifying no match.

Also, the command

```
??
```

uses the remembered search argument.

Expert Editing

This section describes the most advanced **ed** commands.

File Processing Commands

Earlier, we discussed the commands

```
ed
```

and:

```
ed filename
```

ed also has file-handling commands that go beyond those already discussed.

If you decide that you were editing the wrong file, or have finished the current file with a **w**, you can begin to edit an entirely new file with the command:

```
e newfile
```

This forgets all the changes that you have made, if any, up to this point since the last **w** command and begins all over again with **newfile**.

The **e** command:

```
e new
```

has the same effect as

```
ed new
```

issued within COHERENT, but is handier because you do not need to exit **ed** and then reenter to edit a new file. Note that the **ed** command **e**, like the **q** command, issues an error message if another file is being edited and you have not stored it since your last change was made. If you immediately repeat the command, **ed** proceeds even if there are unsaved changes. The command

```
E new
```

commands **ed** to edit the new file, whether or not there are unsaved changes.

The **r** command also reads a new file, but adds it to the file being edited instead of using it to replace the current file. This can be handy for copying one file into another one. For example, if you have a manuscript prefix stored in the file **prefix** to include the prefix at the beginning of the file you are editing, type:

```
0r prefix
```

r inserts the file being read after the line number specified; in this case, line 0 means at the beginning of the file. If used without a line number, **r** appends the newly read lines to the end of the file.

The **w** command writes the entire file if no line number is specified; however, you can line numbers. For example

```
'1,3w new
```

writes the first three lines to file **new**. If the file name is omitted, the lines are written to the remembered file name.

The **w** command is unique in that it never changes the current line. This is true regardless of what line numbers are specified in the range for the command, or how those line numbers were developed.

The **W** command resembles the **w** command, except that it appends lines to the end of the file, whereas **w** creates a new file and erases any previous contents.

The **f** command prints the remembered file name that was set in

```
ed filename
```

or

```
e filename
```

or

```
w filename
```

commands. You can also use **f** to reset the remembered name, by typing:

```
f newname
```

This form of the command tells you what the new remembered file name is, even though you just typed it in.

Note that the command

```
w filename
```

changes the remembered name only if there is currently no remembered name, as does the

r command.

Patterns

Earlier, you were cautioned that certain punctuation characters have special effect in search and substitute commands. These characters are:

```
[ ^ $ * . \ &
```

They are used to form powerful substitute and locator commands. An orderly combination of these special characters is called a *pattern*, sometimes called a *regular expression*. You can use a pattern to find or *match* a variety of strings with one search argument.

The simplest patterns use alphabetic characters and numeric digits, which match themselves. For example,

```
/ab/
```

finds and prints the next line containing the string **ab**.

The next simplest character to use in a pattern is the period or dot. It matches any character except the **newline** character that separates lines. Two periods in succession match any two consecutive characters, and so on. For example, if you have a file that contains algebraic statements of the form

```
a+b
c+e
a-b
a/b
d*e
```

and wanted to find and print any line involving **a** and **b** (in that order), then use the search statement:

```
/a.b/
```

The `.` in this example matches `+`, `-`, and `/`.

Then, you ask, how do I find a string that contains a period? For example, if you want to find all the sentences that ended with "lost." (that is, the word **lost** followed by a period), you might first try:

```
/lost./p
```

This, however, also matches the string "lost " (the word **lost** followed by a space), which is not what you want.

This is where the special character backslash comes in handy. A backslash tells **ed** to treat the next character as a regular character, even if it usually is a special character. Thus, to find "lost.", you need only type:

```
/lost\./p
```

This will not incorrectly find "lost ". If you want to find backslashes in your file, simply say:

```
/\\/p
```

Matching Many With One Character

The asterisk * matches an indefinite number of characters. For example, to remove extra spaces between words in a document, type

```
g/##*/s//#/p
```

(The character # has been substituted here for the space character to make the example more readable.) This replaces each series of spaces by one space.

Note that there are two spaces before the * in the search string. This is necessary because the * matches any length of string, including zero. Therefore, searching for a space followed by any number of spaces finds strings that are at least one space long.

The * matches the longest possible string of the previous character. This requires careful attention on your part, because the string matched by * might be longer than your required string, or even zero in length. Either way could give you unexpected results.

If you have a line

```
a+b-c
```

in your file and want to change it to

```
a+c
```

type the command:

```
s/a.*c/a+c/p
```

However, if the line read instead

```
a+b-c*d+c
```

and you applied the command, the result would be

```
a+c
```

since the .* matches the longest string between any **a** and any **c**.

Beginning and Ending of Lines

The characters ^ and $ match, respectively, the beginning and ending of a line. Thus, you can find and print all lines that end with a bang:

```
g/bang$/p
```

or those that begin with a whimper:

```
g/^whimper/p
```

These two characters can also help you find lines of specific length. If you need to see all lines exactly five characters long, the command

```
g/^.....$/p
```

does the trick. To find and delete all blank lines, type:

```
g/^ *$/d
```

Note that this time the * matches a string of zero spaces. However, this is correct, because a blank line includes lines that have nothing in them, as well as lines that contain only

spaces.

Replacing Matched Part

In many cases of substituting, you find yourself extending a word, or adding information to the end of a phrase. This can lead to extensive retyping of characters. The special **&** character can help out.

This character is special only when used in the right part, or *pattern2* of the substitute command. It means "the string that matched the left part". For example, to add **ing** to the word **help** in the current line, use:

```
s/help/&ing/
```

The ampersand may appear more than once in the right side.

This can be more interesting if the left part has a non-trivial *pattern*. For every word in a line that is preceded by two or more spaces, double the number of spaces before it:

```
s/###*/&&/gp
```

(Again, spaces have been replaced with # for clarity.)

Replacing Parts of Matched String

A more sophisticated feature, which is similar to the ampersand, helps you to rearrange parts of a line. To see how this works, create a file by typing:

```
ed
a
first part=second part
.
w eql
q
```

Two special bracket symbols, \(and \) can be used to delineate patterns in the left part of a substitution expression. Then, you can use the special symbols **\1**, **\2**, etc., to insert the delimited parts. The symbol \(marks the beginnning of the pattern, and \) marks the end. For example, to delete everything in the line except the characters to the left of the =, type

```
ed eql
s/^\(.*\)=.*/\1/p
wq
.
```

In the substitute command, the ^ matches the beginning of the line, **.*** matches "first part", and **=.*** matches the rest of the line. The symbol **\1** signifies the matched characters between the first \((the only one in this example) and \). The **p** then prints the result, which will be:

```
first part
```

With this example, you can interchange parts of a line:

```
ed
a
first part=second part
.
w eql2
q
```

To interchange the two parts, type

```
ed eql2
s/\(.*\)=\(.*\)/\2=\1/
p
wq
```

The result is

```
second part=first part
```

The first portion of the substitution expression,

```
\(.*\)=\(.*\)
```

can be thought of as being in three parts. The first part

```
\(.*\)
```

matches all characters up to but not including the =, which are

```
first part
```

The second part

```
=
```

matches the = in the line, and finally the third part

```
\(.*\)
```

matches all characters following the "=", or

```
second part
```

The remainder of the substitution expression

```
\2=\1
```

which is the replacement part, rebuilds the line in interchanged order. The symbol **\2** replaces the matched string enclosed in the second pair of \(\) delimiters, and the symbol **\1** inserts the matched string enclosed in the first pair of \(\).

The right side of the substitution inserts the second matched expression (from **\2**), then inserts the = sign again, followed finally with the first part of the line from **\1**.

This may appear involved, but can be immensely valuable in situations that require rearrangement of a large number of lines.

The next special characters for patterns that we will consider are the bracket characters [and]. These are used to define the character class. Inside the brackets, put a group of characters; **ed** will match any of them if it appears. For example, to print a line that contains any odd digit, say:

```
g/[13579]/p
```

For even more power and flexibility, you can combine character classes with the asterisk. For example, the following command finds and prints all lines that contain a negative number followed by a period:

```
g/-[0123456789]*\./p
```

This matches lines containing the following example strings:

```
-1.
-666.
-3.7.77
```

You can also match all lower-case letters by listing them in brackets, but the following abbreviation simplifies this:

```
g/[a-z]/p
```

This can also be used for the negative number example above:

```
g/-[0-9]*\./p
```

Most special characters lose their original meaning within the brackets, but one of the special characters, **caret** ^, gets a new meaning. In this context, it matches all characters *except* those listed in the brackets. For example, the following pattern matches a string that begins with **K** and continues with any character except a number:

```
/K[^0-9]/
```

This matches:

```
KQ
KK
KK9
```

but not:

```
K7
kK0
```

Other special characters may be part of a character class, and but lose their special meaning when used in that context. Remember, however, that if you want to match the right bracket, it must appear first in the list. So, to find all occurrences of special characters in the file, type:

```
g/[]^\.*[&]/p
```

Listing Funny Lines

The **p** command prints lines with graphic characters in them. It also prints lines with non-graphic (or *control*) characters, but these do not appear on the screen. For example, printing a line that contains the BEL character **<ctrl-G>** will ring your terminal's bell, but you will not see where the BEL character occurs within the line.

The **l** command behaves like the **p** command, except that it also decodes and prints control characters. For example, if you use the **l** command to print a line that containing the word **bell** followed by a BEL character, you would see:

```
bell\007
```

Note that "007" is the ASCII value for **<ctrl-G>**. (ASCII is the system of encoding characters within your computer; see **ASCII** in the Lexicon for the full ASCII table.) The **l** command displays the backspace character **<ctrl-H>** as a hyphen - overstruck with a <, which appears simply as < on your screen. It displays a tab character as a - overstruck with a >, which appears as a >. If the line being listed with **l** is too long to be displayed on one line on your screen, **l** separates it into two lines, with the backslash character placed at the end of the first line to indicate the split.

All other features of the **p** command apply to the **l** command.

Keeping Track of Current Line

The most commonly used abbreviation in **ed** is the dot, or period, which stands for the current line. Many commands can change the value of the dot, and it is useful to you to be able to anticipate this change when using the abbreviation.

Different classes of commands affect the value of the dot in different ways; in general, however, the simple explanation is usually correct: the current line is the last line processed by the last command to be executed.

Consider, for example, how the substitution command **s** changes the current line:

```
1,$s/flow/change/
p
```

In this example, the current line will be the last line modified by the substitutions; and that will be the line that the **p** command prints.

The **w** command is an exception to this rule. It does not change the current line, regardless of any line range selection or how these ranges are developed.

The **r** command changes the current line to the last of the lines read.

The **d** command sets the current line to the line after the last line deleted unless the last line in the file was deleted, in which case the new last line becomes the current line.

The line insertion commands **i**, **c**, and **a** all leave the current line as the last line added. If no lines are added, however, their behaviors differ: **i** and **c** effectively back up the last line by one, whereas **a** leaves it the same.

When Current Line Is Changed

When the current line changes is also important. Normally, the current line does not change until the command is completed.

To illustrate, create a file **semi** by typing:

```
ed
a
begin
second
first
in between
second
last
.
w semi
q
```

Now, edit the file and type all lines from **first** to **second**:

```
ed semi
/first/,/second/p
Q
```

This will cause an error! The reason is that the search command begins with current line set to **$**, so "first" is found on line 3. But the search for "second" also begins with the current line set at **$**, and finds "second" on line 2. Thus, the command translates to

```
3,2p
```

which is clearly invalid.

To do what was intended, use the **semicolon ;** instead of the comma to separate the two searches. This forces **ed** to change the current line to be changed after the search for **first** rather than after the entire command. Thus, the commands

```
ed semi
/first/;/second/p
Q
```

are correct and will do what is intended. The result will be:

```
first
in between
second
```

The search for **first** still begins with the current line set at **$**. However, after it finds **first**, **ed** resets the current line is set to 3, and begins the search for **second** there, and succeeds on line 5.

Finally, to be sure of where the current line is, you can use the **p** command to show you the line; or you can have **ed** tell you the number of the current line by typing:

```
.=
```

To give you a perspective on where you are with respect to the end of the file, type

```
&=
```

and **ed** will tell you the number of the last line in the file.

You can put any line number expression before = and it will type the result. For example

```
/next/=
```

types the number of the next line to contain "next" (if there is one). The command = never changes the line number.

More About Global Commands

All the global commands discussed thus far have been followed by only one command substitute, print, and delete. You can, however, put several commands after a global command, and execute each of those commands for each line that matches.

To change all occurrences of the word **cacophonous** to the word **noisy** and print the three lines that follow, issue the command:

```
g/cacophonous/s//noisy/\
.+1,.+3p
```

Here, the additional commands are separated by the backslash before the **<return>**. Several commands can be added, and all but the last need the backslash at the end.

This will work for the line-adding commands, as well. To insert a spelling warning before each line that contains the word **occurrance**, issue the command:

```
g/occurrance/i\
((the following line needs spelling check))\
.
```

Note that the last line of the **i** group can be entered without a backslash, in which case the line containing only the period must be omitted. This has the same effect as:

```
g/occurrance/i\
((the following line needs spelling check))
```

You should not depend upon the setting of the current line in any multiline global command. There are two reasons for this. First, if one of the commands is a substitute and the string is not found in the matched line, the current line will not be changed.

Second, the global command operates in two phases. The first part scans the file for lines that match the string argument. **ed** marks these lines internally in a manner similar to the **k** command. The second phase then executes the commands on each of the marked lines. Therefore, you cannot count on a string search following the **g** to set the current line number.

Again, the **v** command behaves in the same way, except that it selects lines that do *not* match the pattern.

Caution is advised when using remembered search arguments, for a similar reason. A search argument is remembered only if the search has been executed. Thus, in a command of the form

```
g/backup/s//reverse/\
s/backin /backing/
```

the first remembered search may use **backup** on some occasion, and "**backin**" on others. The reason for this is that the second phase of the **g** command begins with a remembered search argument of **backup**. After the second line of the multiline command executes, the remembered search argument is be "**backin** ". This remains throughout the remainder of the second **g** phase.

Thus, it is recommended that you avoid remembered search arguments when using multiline global commands.

Issuing COHERENT Commands Within ed

While you are using **ed**, you can issue COHERENT commands by prefixing them with the **!** command.

This can be useful if, for example, you need to discover a file name while in the middle of an edit, and you want to find it without leaving **ed**. Thus, to list your directory while in **ed**, type:

```
!lc
```

ed sends the command to COHERENT and echoes a **!** character when the command is finished.

There is no limitation on the type of command that you may issue with this feature. It is even plausible that you want to start another **ed**.

For More Information

The Lexicon article on **ed** summarizes its commands and options. The COHERENT system also includes two other useful editors: **sed**, the stream editor; and MicroEMACS, the screen editor. Each is introduced with its own tutorial and is summarized in the Lexicon.

Section 7:

Introduction to lex, the Lexical Analyzer

Many computer applications involve reading text strings. This is especially true for man-machine communication.

For some forms of textual input, a programmer can design a program by hand to process it. However, it is much easier to implement such programs when you use a software tool that will automatically construct a program to process the data. The COHERENT command **lex** is such a tool.

lex accepts expressions that describe the text input, and generates a program to process it. In computer-ese, **lex** is a "lexical scanner program generator".

This document tells you how to use **lex**. It presents many simple examples to illustrate how to use its features and how to use the generated program with other tools provided with COHERENT, notably the parser generator **yacc**.

Readers of this document are presumed to be familiar with the C programming language and the use of the COHERENT system. Related documents include *Using the COHERENT System* and the tutorial to yacc, the COHERENT parser generator.

How To Use lex

lex generate lexical scanners for compilers, to do statistical analysis of text, and to generate filters for many diverse tasks. This section gives examples of how to use **lex**. Later sections discuss the concepts used in these examples in detail.

Translating Strings

The first example tells **lex** to match an input string and replace it with a different string; strings not recognized by the program are output unchanged. Enter the following program into the file **rmv.lex**.

```
%%
removeable    printf ("executable");
```

This creates the **lex** specification. Use the following command line to Pass this specification through **lex**:

```
lex rmv.lex
```

This produces a C program named **lex.yy.c**, which you can compile by typing:

```
cc lex.yy.c -ll -o rmv
```

The executable program **rmv** is now ready to use. To illustrate its use, type:

```
rmv
Is this file removeable?
<ctrl-D>
```

Is this file executable?

rmv replies:

```
Is this file executable?
```

Note that the generated program reads from standard input and writes to standard output.

Remove Blanks From Input

The next example deletes all blanks and tabs from the input. Type the following **lex** program into file **nosp.lex**:

```
%%
[ \t]+  ;
```

Generate and compile the program with the following commands:

```
lex nosp.lex
cc lex.yy.c -ll -o nosp
```

To invoke the program, type **nosp**. Now, test it by typing the following:

```
This may be hard to read after processing.
<ctrl-D>
```

nosp outputs:

```
Thismaybehardtoreadafterprocessing.
```

Trimming Blanks

The previous example can be rewritten to remove strings of blanks or tabs and replace them with one space. Type the following into file **onesp.lex**:

```
%%
[ \t]+  printf (" ");
```

Generate and compile this with the following commands:

```
lex onesp.lex
cc lex.yy.c -ll -o onesp
```

Invoke your new program with the command **onesp**. Now, type the following text to test the program:

```
This  should  be  easier  to  read.
<ctrl-D>
```

The words in this input are separated by two spaces. **onesp** prints the following:

```
This should be easier to read.
```

lex Specification Form

This section discusses the form of the **lex** specification.

Simple Form

The examples shown above use the simplest form of a **lex** program. Consider the text of the example **rmv.lex**:

```
%%
removeable    printf ("removable");
```

The symbol

```
%%
```

divides sections of the **lex** specification. Not all specifications need to be present, but at least one **%%** must appear in a **lex** program.

This symbol separates **lex** *definitions* from *rules*. With nothing before the **%%**, there are no definitions. Rules follow the **%%**. No definitions are needed in the simplest of **lex** specifications.

Rules in lex

The format of a **lex** rule is simple. Every rule has two parts. Refer to the program **rmv**:

```
removeable    printf ("removable");
```

The first part begins at the beginning of the line and ends with a space or tab. In the example rule, the first part is

```
removeable
```

This part is called the *pattern*.

The second part follows the space or tab, and is called the *action*. The action in this example is:

```
printf ("removable");
```

When the pattern specified by the rule is found in the input, the corresponding action is performed. Thus, this rule detects every appearance of *removeable* and outputs the correct spelling.

A **lex** program tries each rule's pattern in turn, and performs the associated action if and only if the pattern matches. Actions often modify the input that matched the pattern; they may also do nothing for certain patterns. To illustrate this, type the following specification into file **erase.lex**:

```
%%
erase  ;
```

Then compile the generated program with the following commands:

```
lex erase.lex
cc lex.yy.c -ll -o erase
```

This program copies all its input to its output, except for any appearance of the string **erase**. Invoke the program by typing **erase**, and then test it by typing:

```
Have you erased the blackboard?
<ctrl-D>
```

erase then prints:

```
Have you d the blackboard?
```

If the input contains patterns that do not match any of the patterns in the suite of rules you typed into **lex**, they are simply output unchanged. Usually, you will want to write a rule to cover every case.

Statements in lex

As noted earlier, **lex** is a program generator. It reads the specifications that you prepare for it, and writes a C program that is used with the **lex** library. Many of the actions in the rules you specify, such as

```
printf ("removable");
```

are themselves C statements. These statements are included in the resulting program, along with other statements that **lex** provides so the program can run.

You can include other statements, should the program need them, by placing them in appropriate places. The following program, called **count.lex**, shows how this is done. It counts the number of *tokens*, or strings of non-blank characters. Type the following into the file **count.lex**

```
        int count;
%%
[^ \t\n]+     count++;
[ \t\n]+      ;
%%
yywrap ()
{
        printf ("Number of tokens:%d\n", count);
        return (1);
}
```

Statements other than rule actions appear in two places in the program. The first such statement is in the definition section, which precedes the rule section delimiter **%%**:

```
        int count;
```

This C statement declares the variable **count** to be an integer variable. Notice that it is preceded by a tab; a tab or a space indicates to **lex** that an input line is not a rule.

The second kind of non-rule statement follows the second **%%**, which marks the end of the rules section. **lex** regards anything that follows the second delimiter as being source statements.

The above example includes a function named **yywrap**. **lex** programs always call this function at the end of processing. The above program fills this function with code that prints the number of tokens in the text.

Compile the program by typing the following commands:

```
lex count.lex
cc lex.yy.c -ll -o count
```

Run the program by typing:

```
count <count.lex
```

This counts the tokens in the **count.lex** file itself. **count** will print the following:

```
Number of tokens:21
```

If you do not include a routine named **yywrap**, **lex** will use a standard one.

Groups of Statements

In previous examples, the C statement in the action part of the rule is a single statement. In many **lex** applications, however, you will need to use more than one statement per rule.

To do so, enclose the statements in the braces { }. The following example illustrates grouping. This **lex** specification generates a program to add numbers found in the input and print the total whenever it reads asterisk '*'. Type the following program into **nsum.lex**:

```
        int number, sum;
%%
[0-9]+ {
       sscanf (yytext, "%d", &number);
       sum += number;
       printf ("%s", yytext);
       }
"*"    {
       printf ("%s", yytext);
       printf ("%d", sum);
       sum = 0;
       }
```

To run the generated **nsum** program, enter a sample data file by typing

```
cat >numbers
one two three
1 2 3 4 * 1 2 3 5 *
*
done
<ctrl-D>
```

This builds a sample data file. Run the program by typing:

```
nsum <numbers
```

nsum will print:

```
one two three
1 2 3 4 *10 1 2 3 5 *11
*0
done
```

The statements that follow the definitions

```
[0-9]+
```

and

```
*
```

are enclosed in braces, because each action triggers several statements. Consider the first of these:

```
[0-9]+ {
        sscanf (yytext, "%d", &number);
        sum += number;
        printf ("%s", yytext);
        }
```

The pattern looks for strings of digits. **sscanf** converts each such string into a number and saves it in the variable **number**. Now, consider the second rule:

```
"*"     {
        printf ("%s", yytext);
        printf ("%d", sum);
        sum = 0;
        }
```

This specifies that upon detection of * in the input, the program is to print the sum of the numbers and then reset the counter to zero. In both of these rules, the statement

```
printf ("%s", yytext);
```

prints the number or * so that the output shows the input as well as the total. **lex** defines the variable **yytext**. It always contains the string that matches the rule.

If the input is neither a number or an asterisk, no rule specifically matches it. Therefore, the program echoes it unchanged to the standard output.

Using the Same Action

To make it easier for you to write actions, **lex** allows you *abbreviate* rules; that is, you have to write only once any action that is performed by several rules. To abbreviate rules represented symbolically by

```
p1      action1
p2      action1
```

use the vertical bar operator:

```
p1      |
p2      action1
```

The vertical bar means "use the action from the rule that follows."

Patterns

The first part of each rule in the **lex** rules section is a pattern may that match parts of the input. This section describes how to construct these patterns, sometimes called *regular expressions*. If you are familiar with **ed** and how its patterns work, this will be familiar to you.

Simple Patterns

The simplest kind of pattern is a string of characters that matches itself. A previous section presented an illustration of this:

```
%%
removeable    printf ("executable");
```

This regular expression matches all occurrences of *removeable* that appear in the input text.

Certain characters have special meaning to **lex** patterns. To match a special character literally, you must *quote* it. For example, * has special meaning. To match the asterisk literally (that is to match any '*'s that appear in the input), surround it with quotation marks:

```
"*"
```

Another way to quote characters is to precede it with the backslash character '\'.

```
\*
```

The following characters each have special meaning and must be quoted to be matched as text characters:

```
" \ ( ) < > { } % * + ? [ ] - ^ / $ . |
```

However, within ", the \ still has its meaning, so to match the string * use the regular expression:

```
"\\*"
```

Also, to match a quote character, use:

```
\"
```

Classes of Characters

The power of patterns comes from special characters that match more than one character. The following examines each special character in detail.

The *period* or *dot* matches any character except newline. The following regular expression matches any pair of characters that begins with **J**:

```
J.
```

The following example prints in square brackets any sequence of five characters that ends with a blank. Type the following into the file **five.lex**:

```
%%
....." "      printf ("[%s]", yytext);
```

Compile the program with the following commands:

```
lex five.lex
cc lex.yy.c -ll -o five
```

Invoke it by typing **five**, and test it with the following text:

```
how well does this work?
no match
<ctrl-D>
```

The result is

```
how[ well ]does[ this ]work?
no match
```

The second line of the input does not have any matches. Because the **dot** pattern character does not match the end-of-line character, all five characters that precede the blank must be on the same line.

Another way to match many characters, but selectively, is with the *character class* operation. Enclose in square brackets the set of characters to be matched. Any of the characters listed there will match one character of the input. For example,

```
[0123456789]
```

matches any decimal digit in the input. Characters may be in any order within the brackets. Thus

```
[0246813579]
```

is equivalent to the example above.

To simplify specifying for character classes, you can specify ranges of characters. The beginning and end of the range is separated by a hyphen. To match all decimal digits as above, use:

```
[0-9]
```

To match all alphabetic characters, type:

```
[a-zA-Z]
```

The special character ^, when used after the opening bracket '[', tells **lex** to match any character *except* those enclosed. The following example finds all two-digit numbers not followed by a period or alphabetic character and prints them surrounded by { and }. Type the following into file **twodig.lex**:

```
%%
[0-9][0-9][^\.a-zA-Z]      printf ("{%s}", yytext);
```

Process and compile the program by typing the following commands:

```
lex twodig.lex
cc lex.yy.c -ll -o twodig
```

Invoke the program by typing **twodig**, and test it by entering the following text:

```
12. 12 12a 1 12 b
<ctrl-D>
```

twodig prints the following in reply:

```
12. {12 }12a 1 {12 }b
```

Repetition

In the patterns shown so far, each character matches only one character at a time. However, many interesting input patterns involve repetition of characters.

To match one or more instances of a character, follow it with the pattern operator **+**. Consider the summation example in **nsum.lex**, shown earlier, which recognized strings of input numbers and added them to a total:

```
[0-9]+ {
       sscanf (yytext, "%d", &number);
       sum += number;
       printf ("%s", yytext);
       }
```

The pattern

```
[0-9]+
```

matches a string of one or more digits.

The operator * will match *zero* or more characters of a specified type. The following example deletes all characters between square brackets. Type it into file **star.lex**:

```
%%
\[.*\] printf ("[]");
```

Type the following commands to generate and compile the program:

```
lex star.lex
cc lex.yy.c -ll -o star
```

Invoke the program by typing **star**, and test it by typing the following text:

```
[This should disappear]
[what happens with two] of them [on a line?]
<ctrl-D>
```

A backslash precedes each bracket, because the bracket has a special meaning in regular expressions. The output from this example is:

```
[]
[]
```

In looking at the example's input, you might have expected the output to be:

```
[]
[] of them []
```

lex does not produce the latter output because it generates recognizers that find the longest match if several matches are possible. Therefore, **star** matched the first [, then all characters up to and including the second]. When you write a pattern that matches many characters, you should bear this possibility in mind.

To change the program to match the first], rewrite it as follows:

```
%%
\[[^\]]*\]    printf ("[]");
```

The regular expression now matches a string of all characters except a], when that string is enclosed in square brackets.

The '?' character signals that the previous character or regular expression is optional. In other words, '?' signals zero or one instance of a character or regular expression.

To see how this would be used in a program, consider a text processor that regards a word as being a strings of alphabetic characters that may or may not be followed by a period. The following example does this, and encloses the recognized words in parentheses. Enter it into file **word.lex**:

```
%%
[a-zA-Z]+\.? printf ("(%s)", yytext);
```

Generate and compile the program with the following commands:

```
lex word.lex
cc lex.yy.c -ll -o word
```

Invoke the program by typing **word**, and test it the program with the following text:

```
These are words.
Question mark not included?
<ctrl-D>
```

The result is

```
(These) (are) (words.)
(Question) (mark) (not) (included)?
```

The question mark, like the * and + operators, can also follow another specification of a pattern. If you wanted to able to end a sentence with a character other than a period, the following code will do the job for you:

```
[a-zA-Z]+[.?!,]?
```

The characters

```
.?!,
```

are optional.

The '+' and '*' operators may match many characters. If you wish to match a specific number of characters or patterns, follow the patterns with the repetition within braces { and }. For example

```
[0-9]{3}
```

matches a string of exactly three digits.

You can also specify a range of counts. To match from seven to nine occurrences of lower-case alphabetic characters, use:

```
[a-z]{7,9}
```

Choices and Grouping

To indicate alternate choices of characters or regular expressions, separate them in the regular expression with a vertical bar operator **|**. For example, if you wish to match either three decimal digits or the character **a**, use:

```
[0-9]{3}|a
```

Parentheses help to group the parts of the pattern that are separated by the vertical bar:

```
(abc)|(def)
```

This pattern will match either the string **abc** or the string **def**.

Matching Non-Graphic Characters

Non-special, graphic characters in patterns match themselves. Most non-graphic characters, such as space, tab, and control characters, cannot be matched directly. **lex** provides special sequences to match control characters. The following example removes tabs and blanks from the beginning and end of input lines. Type it into file **deblank.lex**:

```
%%
[ \t]+\n      printf ("\n");
\n[ \t]+      printf ("\n");
```

Generate and compile the program with the following commands:

```
lex deblank.lex
cc lex.yy.c -ll -o deblank
```

Invoke the program by typing **deblank**, and test it by typing the following input:

```
begins with no space or tab
      begins with tab
   begins with three spaces
<ctrl-D>
```

The result will be

```
begins with no space or tab
begins with tab
begins with three spaces
```

The special regular expression **\t** represents *tab*, and **\n** represents *newline*.

To match the backspace character, use **\b**. Form feed is matched by **\f**. To match an arbitrary character with a known octal value, use three octal digits after the backslash; for example,

```
\007
```

More Patterns

This section discusses more advanced capabilities of patterns.

Line Context

Like **ed**, **lex** patterns can include characters that represent the beginning and end of line. To match a line that contains exactly five alphabetic characters, type:

```
^[a-zA-Z]{5}$
```

The character ^ matches the beginning of the line, and **$** matches the end of the line.

Context Matching

A slash (virgule) '/' shows that a following context is necessary to match a string. For example, the following program matches the string **match** only if it is immediately followed by the string **ing**. Type it into file **match.lex**:

```
%%
match/ing     printf ("{%s}", yytext);
```

To compile the program, type the following commands:

```
lex match.lex
cc lex.yy.c -ll -o match
```

To invoke the program, type **match**; and test it by typing the following input:

```
Will this match?
This is a matching test.
<ctrl-D>
```

The result will be

```
Will this match?
This is a {match}ing test.
```

Notice that the string before the slash is matched. The program does not match the part that follows the slash, even though the string must be there for the first part to be matched. Thus, the regular expression that follows the slash may also be matched on its own. To see how this works, type the following into the file **match2.lex**:

```
%%
match/ing     printf ("{%s}", yytext);
ing           printf ("ed");
```

To compile the program, type the following commands:

```
lex match2.lex
cc lex.yy.c -ll -o match2
```

To invoke the program, type **match2**; then test it by typing the following input:

```
Will this match?
This is a matching test.
You must now sing for your supper.
<ctrl-D>
```

The result will be

```
Will this match?
This is a {match}ed test.
You must now sed for your supper.
```

The context-string that follows the / may be a regular expression. The following example matches the whole-number portion of a decimal fraction. Type it into the file **wholept.lex**:

```
%%
"-"?[0-9]+/"."[0-9]+       printf ("(%s)", yytext);
```

To compile the program, type the following commands:

```
lex wholept.lex
cc lex.yy.c -ll -o wholept
```

Invoke the program by typing **wholept**; then type the following to test it:

```
123 12345 1234.567
<ctrl-D>
```

The result will be:

```
123 12345 (1234).567
```

As you can see, the part of the regular expression

```
"-"?
```

matches an optional leading minus sign. Then

```
[0-9]+
```

matches a string of at least one decimal digit. Then, the following context must match the regular expression

```
"."[0-9]+
```

which matches the fractional part of the number. When it finds a number that matches, it prints the number's whole part enclosed in parentheses.

Macro Abbreviations

lex also provides a macro facility that can substantially simplify the writing of complex regular expressions.

A *macro* is a named body of text. A macro processor simply replaces the name of the macro with the text of the macro.

To illustrate, type following example into file **float.lex**. It recognizes integer and floating point constants according to the C format:

```
d [0-9]+
e [Ee][+-]?[0-9]+
%%
{d}\.          |
{d}\.{d}       |
\.{d}          |
{d}\.{e}       |
\.{d}{e}       |
{d}\.{d}{e}    |
{d}{e}         printf ("F:[%s]", yytext);
```

lex replaces the macro name **e** with the code that matches a string of digits at least one digit long. It replaces the macro name **d** with code that matches the number's exponent. These two are invoked in the manner of

```
{d}
```

within a pattern. To compile the program, type the following commands:

```
lex float.lex
cc lex.yy.c -ll -o float
```

Invoke the program by typing **float**, and then test it by typing the following text:

```
1 1. 1.2 1.e4 1e4
.1e4 e4 .1 . 0 1.2e3
<ctrl-D>
```

The result will be:

```
1 F:[1.] F:[1.2] F:[1.e4] F:[1e4]
F:[.1e4] e4 F:[.1] . 0 F:[1.2e3]
```

Context: Start Rules

Many tasks in lexical processing require the program to be aware of a token's context. **lex** lets you make processing conditional upon previously processed input. This is done by using **start conditions**.

Start conditions are named in the definitions section as follows:

```
%S name1 name2
```

where **name1** and **name2** are names of start conditions. These start conditions are then used by prefixing a pattern with the start condition's name enclosed in angle brackets. For example:

```
<name1>
```

For example, you can use one start condition to control the scanning of comments in a Pascal-like language. The start condition is set by the **lex** statement **BEGIN** when the beginning bracket of the comment is found. The comment is scanned for strings that begin with **$** to signal compiler operation. To see how this works, type the following into the file **comment.lex**:

```
%S CMNT
%%
<CMNT>\$[ler]printf ("Option is %s.\n", yytext);
<CMNT>[^\}]   ;
<CMNT>\}            BEGIN 0;
\{                  BEGIN CMNT;
```

To compile, use the following commands:

```
lex comment.lex
cc lex.yy.c -ll -o comment
```

Now, invoke the program by typing **comment**; and test it by typing the following input:

```
{This is a comment}
{This comment has options $l $e $r}
program
information
<ctrl-D>
```

The result will be:

```
Option is $l.
Option is $e.
Option is $r.

program
information
```

The context start condition is named following **BEGIN** in the action part of the rule. To return to the normal condition, use **0** as the context name.

Separate Contexts

If you wish to perform context-dependent processing that is more complex than that shown in the example above, you will find it convenient to use separate contexts.

The names of the contexts are defined in the definitions sections, after the definitions of any start conditions: For example:

```
%C name name ...
```

The **lex** function **yyswitch** switches to a new context.

The body of the context's rules is preceded in the rules section by:

```
%C name
```

To see how this works, type the following into file **pre.lex**. It is part of a program that recognizes the preprocessor statements in a C program:

```
%C PRE
%%
^#      yyswitch (PRE);
[^#\n]+             printf ("[%s]", yytext);
%C PRE
include.+     |
define.+      {
              printf("{%s}", yytext);
              yyswitch(0);
              }
.+            {
              printf ("{??%s}", yytext);
              yyswitch (0);
              }
```

A # in column 1 signals the beginning of a preprocessor statement. Upon recognizing this condition, this program uses **yyswitch** to activate the context **PRE**.

Within this separate context, individual rules recognize different preprocessor statements; this example includes only two. Each of the rules prints the preprocessor line enclosed in braces { }. In addition, the rules switch back to the original (and unnamed) context by the statement

```
yyswitch (0);
```

To compile and test this program, use the following commands:

```
lex pre.lex
cc lex.yy.c -ll -o pre
pre <lex.yy.c
```

This example uses the function **yyswitch** to return to the original context at the end of each rule in the secondary context. Some applications require a return to the context that was previously in force. To assist in this, **yyswitch** returns the value of the previous context.

To modify the example to switch to the previous context, add a statement to the definitions section to declare a variable to hold the previous context:

```
int prev;
```

Then, when switching, save the current context:

```
prev = yyswitch (NEW);
```

To switch back, use:

```
yyswitch (prev);
```

To summarize, you can specify a match at the beginning and end of input lines. You may need a following context for a match. Macros provide a means of abbreviating elements of patterns. **lex** can qualify some patterns based on a start context, or process entirely separate contexts.

More About Writing Actions

This section discusses predefined **lex** actions and how to use them. It also presents other **lex** routines that are useful in writing actions.

ECHO

Many **lex** actions simply output the matched pattern:

```
[0-9]+ printf ("%s", yytext);
```

This form has been used in the examples because many examples also output additional material, such as enclosing braces, to illustrate the matched token.

lex provides a simpler way to echo the exact token matched:

```
[0-9]+        ECHO;
```

The following example echoes all strings of digits twice, and everything else once. Type it into file **double.lex**:

```
%%
[0-9]+        {ECHO; ECHO;}
[^0-9]+              ECHO;
```

To compile the program, use the commands:

```
lex double.lex
cc lex.yy.c -ll -o double
```

To invoke the program, type **double**; and to test it, type the following text:

```
abcdef 123 1234
<ctrl-D>
```

double will reply:

```
abcdef 123123 12341234
```

Processing Overlapping Strings

The **lex** processing illustrated to this point has been restricted to programs whose rules recognize distinct strings. That is, once any character of a string is matched by a regular expression, it cannot be matched by another.

Some applications require that strings be matched by more than one rule; such multiply-matched strings are called *overlapping strings*. The **lex** action word **REJECT** provides this capability. When **REJECT** appears in a rule, other rules can also match the string. Remember, however, that **lex** programs give precedence to the longest string that matches a regular expression.

The following example determines the number of letter pairs, or *digrams*, in its input. The input is presumed to be lower-case letters. Enter the following into **digram.lex**:

```
        int digram [128] [128];
%%
[a-z][a-z]    {
              digram [yytext [0]] [yytext [1]]++;
              REJECT;
              }
.             ;
\n            ;
%%
yywrap ()
{
    int i1, i2;
    for (i1 = 'a'; i1 <= 'z'; i1++)
       for (i2 = 'a'; i2 <= 'z'; i2++)
           if (digram [i1] [i2] != 0)
              printf ("%d\t%c%c\n",
                  digram [i1] [i2], i1, i2);
}
```

To compile the program, type the commands:

```
lex digram.lex
cc lex.yy.c -ll -o digram
```

To invoke the program, type **digram**; and test it with the following text:

```
this is a test of digrams.
<ctrl-D>
```

The result will be:

```
1      am
1      di
1      es
1      gr
1      hi
1      ig
2      is
1      ms
1      of
1      ra
1      st
1      te
1      th
```

yylex

lex places the actions you provide for the rules in your **lex** program into a C routine named **yylex**.

If you add variable declarations in the definitions section before the first **%%**, **yylex** can access them, as in the example **digram.lex**, shown above. You can also declare variables that are local to **yylex**, if you place the declarations after the rules section delimiter and before the first rule. A tab or space must precede the declaration.

The following program is a different version of **digram.lex**, called **digram2.lex**; it uses such a declaration.

```
        int digram [128] [128];
%%
        int t0, t1;
[a-z][a-z]    {
              t0 = yytext [0];
              t1 = yytext [1];
              digram [t0] [t1]++;
              REJECT;
              }
%%
yywrap ()
{
    int i1, i2;
    for (i1 = 'a'; i1 <= 'z'; i1++)
       for (i2 = 'a'; i2 <= 'z'; i2++)
           if (digram [i1] [i2] != 0)
              printf ("%d\t%c%c\n",
                  digram [i1] [i2], i1, i2);
}
```

Header Section

You can insert additional code at the beginning of the generated program by including such code in the definitions section. An earlier example, **count.lex**, demonstrated how to do this:

```
        int count;
%%
[^ \t\n]+     count++;
[ \t\n]+      ;
%%
yywrap ()
{
    printf ("Number of tokens:%d \n ", count);
    return (1);
}
```

A tab or space character must precede the code you include.

If you wish to insert **include** or any other C preprocessor statement at the beginning of the program, however, a different technique must be used. This stems from the fact that the preprocessor statements must begin at the beginning of the line, and the blank or tab precludes this.

The alternative method to add code to the beginning is as follows:

```
%{
... code ...
%}
```

where the **%** symbols are at the beginning of the line.

Additional Routines

If your version of **yywrap** or any of the rules that you write need other routines, you can include code for them after a second **%%**. (This was where **yywrap** was shown in **digram.lex**.) If you wish to provide your own version of **input** or **output**, you must define it there.

Using lex With yacc

Although **lex** can handle many applications by itself, it is often used with the parser-generator **yacc**. For example, programming-language compilers often have parts generated by both **lex** and **yacc**.

Like **lex**, **yacc** is a program generator. Its programs can recognize input that is structured according to a grammar fed to the **yacc** program generator. In most instances, **yacc**-generated programs require *tokens* as input, instead of individual characters. In the context of a programming language, a token is a variable name or a special character (such as an operator). **lex** is often used with **yacc** because **lex** is especially well suited for partitioning text input into tokens.

A **yacc**-generated program expects a token number as input from the routine **yylex**. **yacc** assigns a unique number, or constant definition, to each unique type of token, and expects **yylex** to return these numbers as input.

For your **lex** program to access these predefined constant definitions for token types, you must include the generated **lex** source in the **yacc** specification.

The following examples process very simple input, to illustrate how to assemble **lex**- and **yacc**-generated programs. To begin, type the following into the file **yacclex.yy**:

```
%token beginning midtok ending
%start simplistic
%%
simplistic : beginning middle ending
                    {printf ("recognized"); };
middle :     midtok;
middle :     middle midtok;
%%
```

When **yacc** processes this program, it produces the file **y.tab.h** that contains the token-name definitions. The following **lex** source reads **y.tab.h** to learn of the constant definitions that **yacc** generated; type it into file **yacclex.lex**:

```
%{
#include "y.tab.h"
%}
%%
"("    return (beginning);
")"    return (ending);
[a-zA-Z]     return (midtok);
```

The symbolic definition of the token names are **beginning**, **ending** and **midtok**.

To compile the programs, type the following commands:

```
yacc yacclex.yy
lex yacclex.lex
cc y.tab.c lex.yy.c -ly -ll -o yacclex
```

Type **yacclex** to invoke the new program; and test by typing the following:

```
(abcdef)
```

The result will be:

```
recognized
```

Summary

lex is a utility that generates lexical analyzers according to a set of specifications that you write. *Lexical analysis* means to read a mass of text, recognize strings within that mass, and react appropriately when each type of string is discovered. With **lex**, you can write programs to perform complex analysis of text simply by describing what analysis you want to perform, without worrying about the messy details of how that analysis is actually performed; thus, **lex** is a fine example of what is nowadays called a "fourth-generation language".

lex is especially well suited to work with the parser-generator **yacc**. By using them together, you can efficiently build command processors and even entire computer languages.

Section 8:

Introduction to the m4 Macro Processor

m4 is a macro processor for the COHERENT system. It is a powerful and flexible text processing tool. You can tell it, with a great degree of generality, to search for macro names and replace them with other strings. Macros can also take arguments.

m4 is useful as a front end for the COHERENT assembler **as**, which has no built-in macro facility. It is also useful for higher-level languages like C, as well as for other applications that require replacement of text.

m4 also has powerful facilities for manipulating files, making decisions conditionally, selecting substrings, and performing arithmetic, so it is useful for processing forms.

The command

m4 [*file* ...]

invokes **m4**. **m4** reads each *file* in the order given on the command line; if no *file* is given, **m4** reads from the standard input. The *file* '-' also indicates the standard input; this allows you to perform interactive input while **m4** is processing files. **m4** reports any *file* that it cannot open, and eliminates it from the input stream.

m4 writes its output to the standard output stream. As with other COHERENT commands, the optional output redirection specification '>*outfile*' on the command line redirects the output into *outfile*.

Definitions and Syntax

m4 reads text one line at a time from its input stream. When it reads a line of text, it scans the line for a macro that you have defined. A legal macro name is a string of alphanumeric characters (letters, digits, underscore '_'), the first of which is not a digit. **m4** recognizes the macro name only if it is surrounded by nonalphanumeric characters (i.e., spaces or newline characters) on both sides.

When **m4** finds a macro, it removes it from the input stream and replaces it with its definition. It then writes the resulting modified text (called *replacement text*), onto the input stream. **m4** then reads another line from the input stream, and continues processing.

Text that is contained within single quotation marks is quoted (i.e., is contained between a grave mark ' on the left and an apostrophe ' on the right). All other text is *unquoted*. **m4** searches only unquoted text for macros.

A *macro call* can be either a macro or a macro immediately followed by a set of arguments:

```
macroname(arg1, ..., argn)
```

A set of arguments must start with a left parenthesis that follows the macro immediately (i.e., no space can come between the macro and the left parenthesis). The entire argument set must be enclosed by balanced, unquoted parentheses: parentheses may appear within the text of an argument, but they must always come in balanced pairs. A single left or right parenthesis may be passed by quoting it, e.g. '(' or ')'.

Arguments are separated commas that are both not inside single quotes or inside an inner set of unquoted parentheses. **m4** strips from each argument all leading unquoted spaces, tabs, and newlines. It processes the text of each argument in the same manner that it processes ordinary text; that is, it removes, evaluates, and replaces any recognized macro calls *before* it stores the argument text for possible use within the replacement text. If you wish to pass a macro name or an entire macro call as an argument, it must be quoted. **m4** stores the values of the first nine arguments for possible use in the replacement text. It processes arguments after the ninth, but throws away the results.

m4 does not search quoted text for macros. Instead, it removes the quotation marks and copies the text to the standard output unchanged. Quotes can be nested; that is, quoted text can contain other blocks of quoted text. **m4** removes only the outermost level of quotation marks each time it reads a piece of quoted text. This aids in delaying macro expansion in text until the second (or later) time the text is read by **m4**.

m4 includes numerous predefined macros, which perform various functions. The remainder of this document describes the predefined macros in detail. The final section is a summary, which contains an alphabetized list and brief description of each predefined macro.

Defining Macros

The macro

```
define('name', 'definition')
```

defines a macro *name* and its replacement text *definition*. **m4** replaces every subsequent unquoted occurrence of *name* with *definition*, as described above. For example, the **m4** input

```
define('her', 'COHERENT')
To know, know, know her
Is to love, love, love her ...
```

produces the output

```
To know, know, know COHERENT
Is to love, love, love COHERENT...
```

name should usually be quoted. If it is not quoted and it is being redefined, **m4** sees its old *definition* as the first argument to **define**, which will not have the intended effect. Similarly, *definition* should be quoted if the macro names that occur in it should not be replaced.

Any legal macro name may be the first argument of a **define**. If you redefine a predefined macro, its original function is lost and cannot be recovered.

As noted above, **m4** recognizes a macro name only if it is surrounded by non-alphanumeric characters. For example,

```
define('her', 'COHERENT')
Coherent software is reliable software.
```

produces the output

```
Coherent software is reliable software.
```

m4 does not recognize the characters **her** in the word **Coherent** as a macro name.

The value of the **define** macro is the null or empty string (the string which contains no characters). In other words, **m4** puts nothing (the null string) back on its input stream when it processes a **define** call.

Like predefined macros, user-defined macros may take arguments. **m4** replaces the string *$n* in the macro definition with the value of the *n*th argument, where *n* is a digit (1 to 9). It replaces $0 with the macro name. If the argument set contains fewer than *n* arguments, **m4** replaces *$n* with the null string. **m4** uses functional notation to specify argument sets. Unlike a normal function, however, an **m4** macro does not require a fixed number of arguments. The same macro may be called with or without an argument set, or with argument sets containing different numbers of arguments.

The following macro concatenates its arguments:

```
define('cat', $1$2$3$4$5$6$7$8$9)
```

Then

```
cat(one, 'two', ''three'', 'four, four ',
    five(also,),,seven)
```

becomes

```
onetwo'three'four, four five(also,)seven
```

A more complex definition is:

```
define('comma', ''$0 (which looks like ',')'')
```

This turns each subsequent unquoted occurrence of

```
comma
```

into

```
comma (which looks like ',')
```

Two sets of quotation marks around the replacement text are necessary. When **m4** reads this call to macro **define**, the resultant argument text is:

```
comma
```

for the *name* and

```
'$0 (which looks like ',')'
```

for the *definition*. When **m4** sees the text

```
comma that is not quoted
```

it evaluates and replaces the now-defined macro name **comma** to produce the text

```
'comma (which looks like ',')' that is not quoted
```

on the *input* stream. Because **comma** appears inside a set of quotation marks, **m4** does not treat it as a macro name. For the same reason, the string ',' also passes through unmodified. The final output is:

```
comma (which looks like ',') that is not quoted
```

When the predefined macro **dumpdef** is used without arguments, it returns the names and definitions of all defined macros. For each macro, it returns its quoted name, a tab character, and then its quoted definition; no definition is given for a predefined macro.

When used with arguments,

```
dumpdef(name)
```

returns the quoted definition of each macro name that is appears as an argument.

The predefined macro

```
undefine('name')
```

removes a macro definition. As noted for **define** above, the argument must be quoted to have the desired effect. **undefine** ignores arguments which are not defined macro names. The value of the **undefine** call is the null string. If a predefined macro is undefined, its original function cannot be recovered.

Input Control

The predefined macro **changequote** changes the quote characters. For example:

```
changequote( {, })
```

makes the quote characters the left and right braces. It also removes the effect of the previously defined quotation characters. Missing arguments default to ' for open quotation and ' for close quotation. Thus, **changequote** without arguments restores the original quote characters ' and '. If the arguments are identical, the nesting ability of quotation marks is temporarily lost. Instead, the first instance of the new quote character turns on quoting and the next instance turns off quoting. The value of the **changequote** call is the null string.

The predefined macro **dnl** (delete to newline) "eats" all characters from the input stream up to and including the next newline and returns the null string. It is particularly useful in a string of **define** macro calls. Although **m4** replaces each **define** by the null string, newlines often separate macro definitions, and **m4** copies the newlines to the output stream unchanged. Two ways of using **dnl** are:

```
define(this, that)dnl
define(something, else)dnl

dnl(define(this, that), define(something, else))
```

The first examples use **dnl** without arguments. The final example uses **dnl** with an argument set, which **m4** processes (performing each **define**) and subsequently ignores. The following section describes an alternative (and generally preferable) method of eliminating extraneous newlines in a sequence of **define** calls.

m4 includes two decision-making macros. The predefined macros with the form above, this call of **ifdef** compares *arg1* and *arg2*, and returns *arg3* if they are equal. Otherwise, it compares *arg4* and *arg5*. It returns *arg6* if they are equal, *arg7* otherwise. If more than seven arguments are present and *arg4* and *arg5* are not equal, **ifelse** compares *arg7* and *arg8*. It returns *arg9* if they are equal and the null string otherwise.

In addition to each *file* specified in the command line, any other accessible file may be included in the input stream with the predefined macro

```
include(file)
```

m4 replaces this macro call on the input stream with the entire contents of the specified *file*. If *file* cannot be accessed, **include** causes a fatal error; **m4** prints an error message and exits. The alternative predefined macro

```
sinclude(file)
```

functions exactly like **include**, except that it does not print an error message and stop processing if *file* is inaccessible.

Output Control

m4 maintains ten output streams, numbered zero through nine. Stream 0 is the standard output, where **m4** normally directs its output. Streams 1 through 9 are temporary files. The predefined macro

```
divert(n)
```

diverts output away from stream 0, appending it instead to stream *n*. Any *n* outside the range 0 to 9 causes output to be thrown away until the next **divert** call. **divert** without any arguments or with a nonnumeric argument is equivalent to **divert(0)**. The value of a **divert** call is the null string.

The preceding section described the use of **dnl** to eliminate extraneous newlines on the output stream when processing a sequence of **define** calls. A more readable method of eliminating the newlines is to precede the definitions with **divert(-1)** and follow them with **divert**. **m4** then diverts the extraneous newlines to the nonexistent stream -1.

The predefined macro

```
undivert(streams)
```

fetches text diverted to one or more temporary streams. It appends the text from the specified *streams* in the given order to the *current* output stream. **m4** does not allow diverted text to be undiverted back to the same stream. **undivert** with no arguments undiverts all diversions in numerical order. The value of **undivert** is the null string; undiverted text is *not* scanned for macro calls, but is simply moved from one place to another. **m4** automatically undiverts all diversions in numerical order to the standard output (stream 0) at the end of processing.

The predefined macro **divnum** returns the current diversion number.

The predefined macro

```
errprint(message)
```

sends the given *message* to the standard error stream. The value of **errprint** is the null string.

String Manipulation

The predefined macro

```
substr(string, start, count)
```

returns a substring of a string of characters. The first argument *string* can be anything. The second argument *start* is a number giving the starting position of the desired substring in *string*. Position 0 is the leftmost character of *string*, position 1 is the next character to the right, and so on. If *start* is negative, the orientation switches to the right. Position -1 is the rightmost character of *string*, position -2 is the character to its left, and so on. The third argument *count* specifies the length and direction of the substring. Zero returns the null string. A positive *count* returns a substring consisting of the character addressed by *start* and *count*-1 characters to the right of it. A negative number does the same thing, but to the left. If *count* is omitted, it is assumed to be of the same sign as *start* and large enough to

extend to the end of *string* in that direction. If *start* is omitted, it is assumed to be 0 if *count* is positive or omitted, or -1 if *count* is negative. For example:

```
define('alpha', 'abcdefghijklmnopqrstuvwxyz')
substr(alpha, , )
```

returns

```
abcdefghijklmnopqrstuvwxyz
```

Here both *start* and *count* are omitted and are therefore assumed to be 0 and 26, respectively.

```
substr(alpha, 0, 6)
substr(alpha, , 6)
```

both return

```
abcdef
```

Similarly,

```
substr(alpha, , -6)
substr(alpha, 21, )
```

both return

```
uvwxyz
```

Finally,

```
substr(alpha, -6, )
substr(alpha, 0, 21)
```

both return

```
abcdefghijklmnopqrstu
```

The predefined macro

```
translit(string, characters, replacements)
```

transliterates single characters within a string. It returns *string* with every occurrence of a character specified in *characters* replaced with the corresponding character from *replacements*. If there is no corresponding character, **translit** simply deletes the character. For example:

```
translit(alpha, aeiouy, *+-=/)
```

returns

```
*bcd+fgh-jklmn=pqrst/vwxz
```

Numeric Manipulation

m4 can simulate variables typical of most programming languages by using **define** as the assignment operator. Whenever the defined macro name appears unquoted, **m4** immediately replaces it by its numeric value.

The predefined macros **incr** and **decr** return their argument incremented or decremented by 1. Thus,

```
define('x', 1234)
incr(x)
```

returns

```
1235
```

incr and **decr** assume an argument which is omitted or not a valid number to be 0.

More generally, the predefined macro

eval(*expression*)

evaluates an integer-value arithmetic *expression* and returns the resulting value. The operators available, in order of decreasing precedence, are:

( )	Parentheses for grouping
+ -	Unary plus, negation
^ **	Exponentiation
* / %	Multiplication, division, modulus
+ -	Addition, subtraction
> < >= <= == !=	Comparisons
!	Logical negation
&& &	Logical and
|| |	Logical or

The comparisons and logical operators return either 0 (false) or 1 (true). **eval** performs all arithmetic in **long** integers. **eval** reports an error if its argument is not a well-formed expression.

The predefined macro

len(*string*)

returns a numeric value corresponding to the length of *string*.

The predefined macro

index(*string*, *pattern*)

returns a numeric value corresponding to the first position where *pattern* appears in *string*. If it does not appear, **index** returns -1. Both *pattern* and *string* may be arbitrary strings of any length.

The following example defines a macro **repeat** that repeats its first argument the number of times specified by its second argument.

```
define('repeat',
    'ifelse(eval($2<=0),1,,'repeat($1,decr($2) )'$1)')
```

The definition is recursive; that is, **repeat** calls itself within its own definition. The entire definition is quoted to defer the evaluation of **ifelse** from when **m4** encounters the definition to when it encounters a **repeat** macro call. Similarly, the recursive **repeat** call is quoted to defer its evaluation within the **ifelse**. **eval** checks if the first argument is less than or equal to 0; if so, it returns 1 (true) and **ifelse** returns the null string. Otherwise, **decr** decrements the count, so each successive recursive call has a smaller second argument, and each call appends a copy of the first argument to the previous result. For example:

```
repeat('Ho! ',3)
```

produces

```
Ho! Ho! Ho!
```

COHERENT System Interface

The predefined macro

```
maketemp(string)
```

creates a unique file name for a temporary file. *string* is a six-character string that is normally initialized to **XXXXXX**; **mktemp** replaces all of the **X**s with a pattern of six numerals that form a unique file name in the directory where temporary files are being written. It is the same as the C library routine **mktemp**. It returns the null string if its argument is less than six characters long.

The predefined macro

```
syscmd(command)
```

performs the given COHERENT *command* and returns the null string. It is the same as the C library routine **system**.

A common use of **syscmd** is to create a file which **m4** subsequently reads with an **include**. For example, to get the output from the COHERENT **date** command:

```
define('tempfile', maketemp(/tmp/m4XXXXXX))
define('get_date',
        'syscmd(date >tempfile)''include(tempfile)')
```

In subsequent input, **m4** replaces each occurrence of **get_date** with the system date information. The definition of **tempfile** is unquoted, so **m4** executes the **maketemp** call only once (when it processes the **define**), and it creates only one temporary file. On the other hand, the definition of **getdate** is quoted, so **m4** executes **syscmd** and **include** to get the current time and date each time it processes a **getdate** call. The temporary file should be removed with

```
syscmd(rm tempfile)
```

at the end of the **m4** program.

The following example is more complex. It defines a macro **save** which appends a macro definition to a file.

```
define('save','syscmd('cat>>$2 <<\#
define('$1','dumpdef('$1')')
#
')')
```

The arguments to **define** are the *name*

```
save
```

and the *definition*

```
syscmd('cat >>$2 <<\#
define('$1','dumpdef('$1')')
#
')
```

A typical call of this macro is:

```
save('sample','defs.m4')
```

which saves the macro definition of **sample** in a COHERENT file **defs.m4** containing macro definitions. When **m4** processes this call, the argument of **syscmd** becomes

```
cat >>defs.m4 <<\#
define('sample',
```

followed by the definition of **sample** returned by **dumpdef**, followed by

```
)
#
```

Then **syscmd** executes the COHERENT **cat** command to append the here document delimited by # to the macro definition file **defs.m4**. The leading # delimiter of the here document is quoted with \ to prevent interpretation by the COHERENT shell. Because **save** uses the character # to delimit the here document, it does not work correctly for macro definitions containing #. For example,

```
save('save','defs.m4')
```

does not work as expected.

Errors

m4 reports all errors to the standard error stream. An error produces a line of the form

```
m4: line: message
```

where *line* is a decimal line number and *message* describes the error. For example, the error message

```
m4:  7:  illegal macro name: ab*c
```

indicates an attempt to **define** a macro with the illegal macro name **ab*c** in line 7 of the input stream.

The following error messages may occur:

```
cannot open file
eval: invalid expression
eval: missing or unknown operator
eval: missing value
illegal macro name: name
out of space
/tmp open error
unexpected EOF
```

The *file* or *name* will be the file name or macro name which caused the error, or **{NULL}** if the required argument is omitted.

m4 does not recognize (and therefore does not report) the most common of **m4** errors, namely invoking recursive macro definitions that never terminate. A simple example is the definition

```
define('recursive', 'recursive')
```

When **m4** subsequently encounters a call of **recursive** in its input stream, it replaces it on the input stream with its definition. Because the definition is another call to **recursive**, **m4** replaces it in turn with its definition; the process never terminates. More complicated examples may involve many macro definitions and may be difficult to discover. If **m4** enters an endless loop, you can terminate it from the keyboard by typing the interrupt character (normally **<ctrl-C>**) or the kill character (normally **<ctrl-\>**). If **m4** enters an endless loop while being run in the background, you can terminate it with the **kill** command.

For More Information

The Lexicon entry for **m4** gives a summary of its functions and options.

Section 9:

The make Programming Discipline

make is a utility that relieves you of the drudgery of building a complex C program.

How Does make Work?

To understand how **make** works, it is first necessary to understand how a C program is built: how COHERENT takes you from the C source code that you write to the executable program that you can run on your computer.

The file of C source code that you write is called a *source module*. When COHERENT compiles a source module, it uses the C code in the source module, plus the code in the header files that the code calls to produce an *object module*. This object module is *not* executable by itself. To create an *executable file*, the object module generated from your source module must be handed to a linker, which links the code in the object module with the appropriate library routines that the object module calls, and adds the appropriate C runtime startup routine.

For example, consider the following C program, called **hello.c**:

```
main()
{
        printf("Hello, world\n");
}
```

When COHERENT compiles the file that contains C code shown above, it generates an object module called **hello.o**. This object module is not executable because it does not contain the code to execute the function **printf**; that code is contained in a library. To create an executable program, you must hand **hello.o** to the linker **ld**, which copies the code for **printf** from a library and into your program, adds the appropriate C runtime startup routine, and writes the executable file called **hello**. This third file, **hello**, is what you can execute on your computer.

The term *dependency* describes the relationship of executable file to object module to source module. The executable program *depends* on the object module, the library, and the C runtime startup. The object module, in turn, depends on the source module and its header files (if any).

A program like **hello** has a simple set of dependencies: the executable file is built from one object module, which in turn is compiled from one source module. If you changed the source module **hello.c**, creating an updated version of **hello** would be easy: you would simply compile **hello.c** to create **hello.o**, which you would link with the library and the runtime startup to create **hello**. COHERENT, in fact, does this for you automatically: all you need to do is type

```
cc hello.c
```

and COHERENT takes care of everything.

On the other hand, the dependencies of a large program can be very complex. For example, the executable file for the MicroEMACS screen editor is built from several dozen object modules, each of which is compiled from a source module plus one or more header files. Updating a program as large as MicroEMACS, even when you change only one source module, can be quite difficult. To rebuild its executable file by hand, you must remember the names of all of the source modules used, compile them, and link them into the executable file. Needless to say, it is very inefficient to recompile several dozen object modules to create an executable when you have changed only one of them.

make automatically rebuilds large programs for you. You prepare a file, called a **makefile**, that describes your program's chain of dependencies. **make** then reads your **makefile**, checks to see which source modules have been updated, recompiles only the ones that have been changed, and then relinks all of the object modules to create a new executable file. **make** both saves you time, because it recompiles only the source modules that have changed, and spares you the drudgery of rebuilding your large program by hand.

Try make

The following example shows how easy it is to use **make**.

To begin, **make** examines the time and date that COHERENT has stamped on each source file and object module. When you edit a source module, COHERENT marks it with the time at which you edited it. Thus, if a source module has a time that is *later* than that of its corresponding object module, then **make** knows that the source module was changed since the object module was last compiled and it will compile a new object module from the altered source module. If you do not reset the time on your system whenever you reboot, *every time*, some files will not have the correct date and time and **make** cannot work correctly.

To see how **make** works, try compiling a program called **factor**. It is built from the following files:

```
atod.c
factor.c
makefile
```

All three are included with your copy of COHERENT.

Use the **cd** command to shift into directory **/usr/src/sample**.

Now, type **make**. **make** will begin by reading **makefile**, which describes all of **factor**'s dependencies. It will then use the **makefile** description to create **factor**. The following will appear on your screen:

```
cc -c factor.c
cc -c atod.c
cc -f -o factor factor.o atod.o -lm
```

Each of these messages describes an action that **make** has performed. The first shows that **make** is compiling **factor.c**, the second shows that it is compiling **atod.c**, and the third shows that it is linking the compiled object modules **atod.o** and **factor.o** to create the executable file **factor**.

When **make** has finished, the COHERENT prompt will return. To see how your newly compiled program works, type

```
factor 100
```

factor will calculate the prime factors of its argument **100**, and print them on the screen.

To see what happens if you try to re-make your file, type **make** again. **make** will run quietly for a moment, and then exit. **make** checked the dates and times of the object modules and their corresponding source modules and saw that the object modules had a time later than that of the source modules. Because no source module changed, there was no need to recompile an object module or relink the executable file, so **make** quietly exited.

To see what happens when one of the source modules changes, try the following. Use the MicroEMACS screen editor to open the file **factor.c** for editing. Insert the following line into the comments at the top, immediately following the **/***:

```
* This comment is for test purposes only.
```

Now exit. Type **make** once again. This time, you will see the following on your screen:

```
cc -O -c factor.c
cc -O -f -o factor factor.o atod.o -lm
```

Because you altered the source module **factor.c**, its time was later than that of its corresponding object module, **factor.o**. When **make** compared the times of **factor.c** and **factor.o**, it noted that **factor.c** had been altered. It then recompiled **factor.c** and relinked **factor.o** and **atod.o** to re-create the executable file **factor**. **make** did not touch the source module **atod.c** because **atod.c** had not been changed since the last time it was compiled.

As you can see, **make** greatly simplifies the construction of a C program that uses more than one source module.

Essential make

Although **make** is a powerful program, its basic features are easy to master. This section will show you how to construct elementary **make** scripts.

The makefile

When you invoke **make**, it searches the directories named in the environmental variable **PATH** for a file called **makefile**. As noted earlier, the **makefile** is a text file that describes a C program's dependencies. It also describes the type of program you wish to build, and the commands for building it.

A **makefile** has three basic parts.

First, the **makefile** describes the executable file's dependencies. That is, it lists the object modules needed to create the executable file. The name of the executable file is always followed by a colon ':' and then by the names of files from which the target file is generated.

For example, if the program **feud** is built from the object modules **hatfield.o** and **mccoy.o**, you would type:

```
feud:    hatfield.o mccoy.o
```

If the files **hatfield.o** and **mccoy.o** do not exist, **make** knows to create them from the source modules **hatfield.c** and **mccoy.c**.

Second, the **makefile** holds one or more *command* lines. The command line gives the command to compile the program in question. The only difference between a **makefile** command line and an ordinary **cc** command is that a **makefile** command line *must* begin with a space or a tab character.

For example, the **makefile** to generate the program **feud** must contain the following command line:

```
cc -o feud hatfield.o mccoy.o
```

For a detailed description of the **cc** command and its options, refer to the entry for **cc** in the Lexicon.

Third, the **makefile** lists all of the header files that your program uses. These are given so that **make** can check if they were modified since your program was last compiled. For example, if the program **hatfield.c** used the header file **shotgun.h** and **mccoy.c** used the header files **rifle.h** and **pistol.h**, the **makefile** to generate **feud** would include the following lines:

```
hatfield.o: shotgun.h
mccoy.o: rifle.h pistol.h
```

Thus, the entire **makefile** to generate the program **feud** is as follows:

```
feud: hatfield.o mccoy.o
      cc -o feud hatfield.o mccoy.o

hatfield.o: shotgun.h
mccoy.o: rifle.h pistol.h
```

A **makefile** may also contain *macro definitions* and *comments*. These are described below.

Building a Simple makefile

The program **factor** is built from two source modules, **factor.c** and **atod.c**. No header files are used. The **makefile** contains the following two lines:

```
factor: factor.o atod.o
      cc -f -o factor factor.o atod.o -lm
```

The first line describes the dependency for the executable file **factor** by naming the two object modules needed to build it. The second line gives the command needed to build **factor**. The option **-lm** at the end of the command line tells **cc** that this program needs the mathematics library **libm** when the program is linked. No header file dependencies are described because these programs use no special header files. (Header files are described by the **#include** preprocessor instruction.)

Comments and Macros

You can embed comments within a **makefile**. A *comment* is a line of text that is ignored; this lets you "document" the file, so that whoever reads it will now know what it is for. **make** ignores all lines that begin with a pound sign '#'. For example, you may wish to include the following information in your **makefile** for **factor**:

```
# This makefile generates the program "factor".
# "factor" consists of the source modules "factor.c" and
# "atod.c".  It uses the standard mathematics library
# "libm", but it requires no special header files.
# "-f" lets you use printf for floating-point numbers.

factor: factor.o atod.o
      cc -f -o factor factor.o atod.o -lm
```

Anyone who reads this file will know immediately what it is for by looking at the comments.

make also lets you define macros within your **makefile**. A *macro* is a symbol that represents a string of text. Usually, a macro is defined at the beginning of the **makefile** using a *macro definition statement*. This statement uses the following syntax:

```
SYMBOL = string of text
```

Thereafter, when you use the symbol in your **makefile**, it must begin with a dollar sign '$' and be enclosed within parentheses.

Macros eliminate the chore of retyping long strings of file names. For example, with the **makefile** for the program **factor**, you may wish to use a macro to substitute for the names of the object modules out of which it is built. This is done as follows:

```
# This makefile generates the program "factor".
# "factor" consists of the source modules "factor.c" and
# "atod.c".  It uses the standard mathematics library
# "libm", but it requires no special header files.
# "-f" lets you use printf for floating-point numbers.

OBJ = factor.o atod.o
factor: $(OBJ)
       cc -o factor $(OBJ) -lm
```

The macro **OBJ** is used in this **makefile**. If you use a macro that has not been defined, **make** substitutes an empty string for it. The use of a macro makes sense when generating large files out of a dozen or more source modules. You avoid retyping the source module names, and potential errors are avoided.

Note that you can define macros in the **makefile**, in the environment, or as a command-line argument. A macro defined as a command-line argument always overrides a definition of the same macro in the environment or in the **makefile**. Normally, a definition in a **makefile** overrides a definition of the same macro name in the environment; however, the **-e** option to **make** forces definitions in the environment to override those in the **makefile**.

Setting the Time

As noted above, **make** checks to see which source modules have been modified before it regenerates your C program. This is done to avoid wasteful recompiling of source modules that have not been updated.

make determines that a source module has been altered by comparing its date against that of the target program. For example, if the object module **factor.o** was generated on March 16, 1987, 10:52:47 A.M., and the source module **factor.c** was modified on March 20, 1987, at 11:19:06 A.M., **make** will know that **factor.c** needs to be recompiled because it is *younger* than **factor.o**.

Building a Large Program

As shown earlier, **make** can ease the task of generating a large program. The following is the **makefile** used to generate the screen editor MicroEMACS:

```
# makefile for "MicroEMACS"

CFLAGS = -O
LFLAGS = /usr/lib/libterm.a
OBJ=ansi.o basic.o buffer.o display.o file.o \
       fileio.o line.o main.o random.o region.o \
       search.o spawn.o termio.o vt52.o window.o \
       word.o tcap.o

me: $(OBJ)
       cc -o me $(OBJ) $(LFLAGS)

$(OBJ):       ed.h
```

The first line is commentary that describes the file.

The next five lines define macros that are used on the target and command line. The first macros will be discussed in the following section. The second macro substitutes for the name of a special library that is needed to create this program. The third macro, which is three lines long, stands for the names of the source modules that produce MicroEMACS. A backslash '\' must be used to tell **make** that the definition is carried over onto the next line.

The next line names the target file (**me**) and the files used to construct it, here represented by the macro **OBJ**.

Next comes the command line, which gives the compilation to be performed. This line *must* begin with a space or a tab.

The last line lists the header file **ed.h**, which is required by all of the files used to generate MicroEMACS.

Command Line Options

Although **make** is controlled by your **makefile**, you can also control **make** by using command line options. These allow you to alter **make**'s activity without having to edit your **makefile**.

Options must follow the command name on the command line and begin with a hyphen, '-', using the following format. The square brackets merely indicate that you can select any of these options; do *not* type the brackets when you use the **make** command:

```
make [ -deinpqrst ] [ -f filename ]
```

Each option is described below.

-d (debug) **make** describes all of its decisions. You can use this to debug your **makefile**.

-e "Environment" option: force definitions in the environment to override those in the **makefile**. For example, if the **makefile** defines

```
foo=makefoo
```

and the environment defines

```
foo=envfoo
```

then **$(foo)** expands to **makefoo** if you use the command **make** but expands to **envfoo** if you use the command **make -e**.

-f *filename*
(file) option tells **make** that its commands are in a file other than **makefile**. For example, the command

```
make -f smith
```

tells **make** to use the file **smith** rather than **makefile**. If you do not use this option, **make** searches the directories named in the environmental variable **PATH**, and then the current directory for a file entitled **makefile** or **Makefile** to execute.

-i (ignore errors) **make** ignores error returns from commands and continues processing. Normally, **make** exits if a command returns an error status.

-n (no execution) **make** tests dependencies and modification times but does not execute commands. This option is especially helpful when constructing or debugging a **makefile**.

-p (print) **make** prints all macro definitions and target descriptions.

-q Return a zero exit status if the targets are up to date. Do not execute any commands.

-r (rules) **make** does not use the default macros and commands from **/usr/lib/makemacros** and **/usr/lib/makeactions**. These files will be described below.

-s (silent) **make** does not print each command line as it is executed.

-t (touch) **make** changes the modification time of each executable file and object module to the current time. This suppresses recreation of the executable file, and recompilation of the object modules. Although this option is used typically after a purely cosmetic change to a source module or after adding a definition to a header file, it must be used with great caution.

Other Command Line Features

In addition to the options listed above, you may include other information on your command line.

First, you can define macros on the command line. A macro definition must *follow* any command line options. Arguments, including spaces, must be surrounded by quotation marks, as spaces are significant to the shell. For example, the command line

```
make -n -f smith "OPT=-DTEST"
```

tells **make** to run in the *no execution* mode, reading the file **smith** instead of **makefile**, and defining the macro **OPT** to mean **-DTEST**.

The ability to define macros on the command line means that you can create a **makefile** using macros that are not yet defined; this greatly increases **make**'s flexibility and makes it even more helpful in creating and debugging large programs. In the above example, you can define a command line as follows:

```
cc $(OPT) example.c
```

When you define the macro **OPT** on the command line, then the program is compiled using the **-DTEST** option, which defines the preprocessor variable **TEST**.

As noted above, a macro defined on the command line always overrides an identically named macro defined either in the environment or in the **makefile**.

Another command-line feature is the ability to change the name of the *target file* on the command line. Normally, the target file is the executable file that you wish to create, although, as will be seen, it does not have to be. As will be discussed below, a **makefile** can name more than one target file. **make** normally assumes that the target is the first target file named in **makefile**. However, the command line may name one or more target files at the end of the line, after any options and any macro definitions.

To see how this works, recall the program **factor** described above. **factor** is generated out of the source modules **factor.c** and **atod.c**. The command

```
make atod.o
```

with the **makefile** outlined above would produce the following **cc** command line:

```
cc -c atod.c
```

if the object module **atod.o** does not exist or is outdated. Here, **make** compiles **atod.c** to create the target specified in the **make** command line, that is, **atod.o**, but it does not create **factor**. This feature allows you to apply your **makefile** to only a portion of your program.

The use of special, or *alternative*, target files is discussed below.

Advanced make

This section describes some of **make**'s advanced features. For most of your work, you will not need these features; however, if you create an extremely complex program, you will find them most helpful.

Default Rules

The operation of **make** is governed by a set of *default rules*. These rules were designed to simplify the compilation of a typical program; however, unusual tasks may require that you bypass or alter the default rules.

To begin, **make** uses information from the files **/usr/lib/makemacros** and **/usr/lib/makeactions** to define default macros and compilation commands. **make** uses the commands in **makemacros** and **makeactions** whenever the **makefile** specifies no explicit regeneration commands. The command line option **-r** tells **make** not to use the macros and actions defined in **makemacros** and **makeactions**.

As shown in earlier examples, **make** knows by default to generate the object module **atod.o** from the source module **atod.c** with the command

```
cc -c atod.c
```

The macro **.SUFFIXES** defines the suffixes **make** knows about by default. Its definition in **makemacros** includes both the **.o** and **.c** suffixes.

make's files **makemacros** and **makeactions** use pre-defined macros to increase their scope and flexibility. These are as follows:

$< This stands for the name of the file or files that cause the action of a default rule. For example, if you altered the file **atod.c** and then invoked **make** to rebuild the executable file **factor**, **$<** would then stand for **atod.c**.

$*** This stands for the name of the target of a default rule with its suffix removed. If it had been used in the above example, **$* would have stood for **atod**.

$<** and **$* work *only* with default rules; these macros will not work in a **makefile**.

$? This stands for the names of the files that cause the action and that are younger than the target file.

$@ This stands for the target name.

You can use the macros **$?** and **$@** in a **makefile**. For example, the following rule updates the archive **libx.a** with the objects defined by macro **$(OBJ)** that are out of date:

```
libx.a:      $(OBJ)
        ar rv libx.a $?
```

makemacros also contains default commands that describe how to build additional kinds of files:

- **AS** and **ASFLAGS** call the *assembler* to assemble **.o** files out of source modules written in assembly language rather than C.
- **YACC** and **YFLAGS** call **yacc** to build **.o** or **.c** files from **.y** files.
- **LEX** and **LFLAGS** call **lex** to build **.o** or **.c** files from **.l** files.

You can change the default rules of **make** by changing them in **makeactions** and changing the definition of any of the macros as given in **makemacros**.

Source File Path

If a file is not specified with an absolute path name beginning with '/', **make** first looks for the file in the current directory. If the file is not found in the current directory, **make** searches for it in the list of directories specified by the macro **$(SRCPATH)**. This allows you to compile a program in an object directory separate from the source path.

For example

```
export SRCPATH=/usr/src/local/me
make
```

or alternatively

```
make SRCPATH=/usr/src/local/me
```

builds objects in the current directory as specified by the **makefile** from sources kept in directory **/usr/src/local/me**. To test changes to a program built from several source files, copy only the files you wish to change to the current directory; **make** will use the local sources and find the other other sources on the **$(SRCPATH)**.

Note that **$(SRCPATH)** can be a single directory, as in the above example, or a list of directories. In the latter case, each entry in the list must be separated by a colon ':', as described in the Lexicon entry for the function **path()**.

Double-Colon Target Lines

An alternative form of target line simplifies the task of maintaining archives. This form uses the double colon "::" instead of a single colon ':' to separate the name of the target from those of the files on which it depends.

A target name can appear on only one single-colon target line, whereas it can appear on several double-colon target lines. The advantage of using the double-colon target lines is that **make** will remake the target by executing the commands (or its default commands) for the *first* such target line for which the target is older than a file on which it depends.

For example, for the program **factor** described earlier, assume that two versions of the source modules **factor.c** and **atod.c** exist: **factora.c** plus **atoda.c**, and **factorb.c** plus **atodb.c** The **makefile** would appear as follows:

```
OBJ1 = factora.o atoda.o
OBJ2 = factorb.o atodb.o

factor:: $(OBJ1)
        cc -c $(OBJ1) -lm

factor:: $(OBJ2)
        cc -c $(OBJ2) -lm
```

This **makefile** tells **make** to do the following: (1) Check if either **factora.o** or **atoda.o** is younger than **factor**. (2) If either one is, regenerate **factor** using this version of these files. (3) If neither **factora.o** nor **atoda.o** is younger than **factor**, then check to see if either **factorb.o** or **atodb.o** is younger than **factor**. (4) If either of them is, then regenerate **factor** using the youngest version of these files.

This technique allows you to maintain multiple versions of source files in the same directory and selectively recompile the most recently updated version without having to edit your **makefile** or otherwise trick the system.

You cannot target a file in both a single-colon and a double-colon target line.

Alternative Uses

make is a program that helps you construct complex things from a number of simpler things.

make usually is used to build complex C programs: the executable file is made from object modules, which are made from source modules and header files. However, **make** can be used to create any type of file that is constructed from one or more source modules. For example, an accountant can use **make** to generate monthly reports from daily inventories: all the accountant has to do is prepare a **makefile** that describes the dependencies (that is, the name of the monthly report they wish to create and the names of the daily inventories from which it is created), and the command required to generate the monthly report. Thereafter, to recreate the report, all the accountant has to do to generate a monthly report is type **make**.

In another example, the **makefile** can trigger program maintenance commands. For example, the target name **backup** might define commands to copy source modules to another directory; typing **make backup** saves a copy of the source modules. Similar uses include removing temporary files, building archives, executing test suites, and printing listings. A **makefile** is a convenient place to keep all the commands used to maintain a program.

The following example shows a **makefile** that defines two special target files, **printall** and **printnew**, to be used with the source files for the program **factor**.

```
# This makefile generates the program "factor".
# "factor" consists of the source modules "factor.c" and
# "atod.c".  It uses the standard mathematics library
# libm, but it requires no special header files.

OBJ = factor.o atod.o
SRC = factor.c atod.c

factor: $(OBJ)
       cc -o factor $(OBJ) -lm

# program to print all the updated source modules
# used to generate the program "factor"

printall:
       pr $(SRC) | lpr
       >printnew

printnew: $(SRC)
       pr $? | lpr
       >printnew
```

In this instance, typing the command

```
make printall
```

forces **make** to generate the target **printall** rather than the target **factor**, which is the default as it appears first in the **makefile**. The **pr** and **lpr** commands are then used to print a listing of all files defined by **SRC**. The macro **OBJ** cannot be used with these commands because it would trigger the printing of the object files, which would not be of much use. It also creates an empty file **prnew**. This new file serves only to record the time the listing is printed. This tactic is performed in order to record the time that the listing was last generated so that **make** will know what files have been updated when you next use **printnew**.

Typing the command

```
make printnew
```

forces **make** to generate the target **printnew** rather than the default target **factor**. **printnew** prints only the files named in the macro **SRC** that have changed since any files were last printed.

Special Targets

A few target names have special meanings to **make**. The name of each special target begins with '.' and contains upper-case letters.

The target name **.DEFAULT** defines the default commands **make** uses if it cannot find any other way to build a target. The special target **.IGNORE** in a **makefile** has the same effect as the **-i** command line option. Similarly, **.SILENT** has the same effect as the **-s** command line option.

Errors

make prints "*command* exited with status *n*" and exits if an executed *command* returns an error status. However, it ignores the error status and continues processing if the **makefile** command line begins with a hyphen '-' or if the **make** command line specifies the **-i** option.

make reports an error status and exits if the user interrupts it. It prints "**can't open** *file*" if it cannot find the specification *file*. It prints "**Target** *file* **is not defined**" or "**Don't know how to make** *target*" if it cannot find an appropriate *file* or commands to generate *target*. Other possible errors include syntax errors in the specification file, macro definition errors, and running out of space. The error messages **make** prints are generally self-explanatory; however, a table of error messages and brief descriptions of them are given in a later section of this manual.

Exit Status

make normally returns a status of zero if it succeeds, and of one if an error occurs. With the **-q** option (described above), **make** returns zero if all files are up to date and two if they are not up to date.

Where To Go From Here

The Lexicon article on **make** summarizes **make**'s options and features. The source code included with the COHERENT system, such as that for the MicroEMACS screen editor, includes **makefiles**. Studying them will show you how **make** has been used to control the building of large, real-world applications.

Section 10:

Introduction to MicroEMACS

This section introduces MicroEMACS, the interactive screen editor for COHERENT.

What is MicroEMACS?

MicroEMACS is an interactive screen editor. An *editor* lets you type text into your computer, name it, store it, and recall it later for editing. *Interactive* means that MicroEMACS accepts an editing command, executes it, displays the results for you immediately, then waits for your next command. *Screen* means that you can use nearly the entire screen of your terminal as a writing surface: you can move your cursor up, down, and around your screen to create or change text, much as you move your pen up, down, and around a piece of paper.

These features, plus the others that will be described in the course of this tutorial, make MicroEMACS powerful yet easy to use. You can use MicroEMACS to create or change computer programs or any type of text file.

This version of MicroEMACS was developed by Mark Williams Company from the public-domain program written by David G. Conroy. This tutorial is based on the descriptions in his essay *MicroEMACS: Reasonable Display Editing in Little Computers*. MicroEMACS is derived from the mainframe display editor EMACS, created by Richard Stallman.

For a summary of MicroEMACS and its commands, see the entry for **me** in the Lexicon.

Keystrokes: <ctrl>, <esc>

The MicroEMACS commands use **control** characters and **meta** characters. Control characters use the *control* key, which is marked **Control** or **ctrl** on your keyboard. Meta characters use the *escape* key, which is marked **Esc**.

Control works like the *shift* key: you hold it down *while* you strike the other key. This tutorial represent it with a hyphen; for example, pressing the control key and the letter 'X' key simultaneously will be shown as follows:

```
<ctrl-X>
```

The **esc** key, on the other hand, works like an ordinary character. You strike it first, *then* strike the letter character you want. This tutorial does *not* represent the *Escape* codes with a hyphen; for example, it represents **escape X** as:

```
<esc>X
```

Becoming Acquainted with MicroEMACS

Now you are ready for a few simple exercises that will help you get a feel for how MicroEMACS works.

To begin, type the following command to COHERENT:

```
me sample
```

Within a few seconds, your screen will have been cleared of writing, the cursor will be positioned in the upper left-hand corner of the screen, and a command line will appear at the bottom of your screen.

Now type the following text. If you make a mistake, just backspace over it and retype the text. Press the carriage return or enter key after each line:

```
main()
{
        printf("Hello, world!\n");
}
```

Notice how the text appeared on the screen character by character as you typed it, much as it would appear on a piece of paper if you were using a typewriter.

Now, type **<ctrl-X><ctrl-S>**; that is, type **<ctrl-X>**, and then type **<ctrl-S>**. It does not matter whether you type capital or lower-case letters. Notice that this message has appeared at the bottom of your screen:

```
[Wrote 4 lines]
```

This command has permanently stored, or *saved*, what you typed into a file named **sample**.

Type the next few commands, which demonstrate some of the tasks that MicroEMACS can perform for you. These commands will be explained in full in the sections that follow; for now, try them to get a feel for how MicroEMACS works.

Type **<esc><**. Be sure that you type a less-than symbol '<'. Notice that the cursor has returned to the upper left-hand corner of the screen. Type **<esc>F**. The cursor has jumped forward by one word, and is now on the left parenthesis.

Type **<ctrl-N>**. Notice that the cursor has jumped to the next line, and is now just to the right of the left brace '{'.

Type **<ctrl-A>**. The cursor has jumped to the *beginning* of the second line of your text.

Type **<ctrl-N>** again. Now the cursor is at the beginning of the third line of the program, the **printf** statement.

Now, type **<ctrl-K>**. The third line of text has disappeared, leaving an empty space. Type **<ctrl-K>** again. The empty space where the third line of text had been has now disappeared.

Type **<esc>>**. Be sure to type a greater-than symbol '>'. The cursor has jumped to the space just below the last line of text. Now type **<ctrl-Y>**. The text that you erased a moment ago has reappeared, but in a new position on he screen.

By now, you should be feeling more at ease with typing MicroEMACS's *control* and *escape* codes. The following sections will explain what these commands mean. For now, exit from MicroEMACS by typing **<ctrl-X><ctrl-C>**, and when the message

```
Quit [y/n]?
```

appears type **y** and then **<return>**. This will return you to COHERENT.

Beginning a Document

This section practices on the file **example1.c**. This file is stored in the directory **/usr/src/example**. Before beginning, copy it into the current directory with this command:

```
cp /usr/src/sample/example1.c .
```

Now, type the following command to invoke MicroEMACS:

```
me example1.c
```

In a moment, the following text will appear on your screen:

```
/*
 * This is a simple C program that computes the results
 * of three different rates of inflation over the
 * span of ten years.  Use this text file to learn
 * how to use MicroEMACS commands
 * to make creating and editing text files quick,
 * efficient and easy.
 */
#include <stdio.h>
main()
{
     int i;                  /* count ten years */
     float w1, w2, w3;       /* three inflated quantities */
     char  *msg = " %2d\t%f %f %f\n";/* printf string */
     i = 0;
     w1 = 1.0;
     w2 = 1.0;
     w3 = 1.0;
     for (i = 1; i<= 10; i++) {
          w1 *= 1.07;        /* apply inflation */
          w2 *= 1.08;
          w3 *= 1.10;
          printf (msg, i, w1, w2, w3);
     }
}
```

When you invoke MicroEMACS, it *copies* that file into memory. Your cursor also moved to the upper left-hand corner of the screen. At the bottom of the screen appears the *status line*, as follows:

```
-- Coherent MicroEMACS -- example1.c -- File: example1.c --
```

The word to the left, MicroEMACS, is the name of the editor. The word in the center, **example1.c**, is the name of the *buffer* that you are using. (We will describe later just what a buffer is and how you use it.) The name to the right is the name of the text file that you are editing.

Moving the Cursor

Now that you have read a text file into memory, you are ready to edit it. The first step is to learn to move the cursor.

Try these commands for yourself as we described them. That way, you will quickly acquire a feel for handling MicroEMACS's commands.

Moving the Cursor Forward

This first set of commands moves the cursor forward:

<ctrl-F>	Move forward one space
<esc>F	Move forward one word
<ctrl-E>	Move to end of line

To see how these commands work, do the following: Type the *forward* command **<ctrl-F>**. As before, it does not matter whether the letter 'F' is upper case or lower case. The cursor has moved one space to the right, and now is over the character '*' in the first line.

Type **<esc>F**. The cursor has moved one *word* to the right, and is now over the space after the word **this**. MicroEMACS considers only alphanumeric characters when it moves from word to word. Therefore, the cursor moved from under the * to the space after the word **this**, rather than to the space after the *. Now type the *end of line* command **<ctrl-E>**. The cursor has jumped to the end of the line and is now just to the right of the **e** of the word **three**.

Moving the Cursor Backwards

The following summarizes the commands for moving the cursor backwards:

<ctrl-B>	Move back one space
<esc>B	Move back one word
<ctrl-A>	Move to beginning of line

To see how these work, first type the *backward* command **<ctrl-B>**. As you can see, the cursor has moved one space to the left, and now is over the letter **e** of the word **three**. Type **<esc>B**. The cursor has moved one *word* to the left and now is over the **t** in **three**. Type **<esc>B** again, and the cursor will be positioned on the **o** in **of**.

Type the *beginning of line* command **<ctrl-A>**. The cursor jumps to the beginnning of the line, and once again is resting over the '/' character in the first line.

From Line to Line

<ctrl-P>	Move to previous line
<ctrl-N>	Move to next line

These two commands move the cursor up and down the screen. Type the *next line* command **<ctrl-N>**. The cursor jumps to the space before the '*' in the next line. Type the *end of line* command **<ctrl-E>**, and the cursor moves to the end of the second line to the right of the period.

Continue to type **<ctrl-N>** until the cursor reaches the bottom of the screen. As you reached the first line in your text, the cursor jumped from its position at the right of the period on the second line to just right of the brace on the last line of the file. When you

move your cursor up or down the screen, MicroEMACS tries to keep it at the same position within each line. If the line to which you are moving the cursor is not long enough to have a character at that position, MicroEMACS moves the cursor to the end of the line.

Now, practice moving the cursor back up the screen. Type the *previous line* command **<ctrl-P>**. When the cursor jumped to the previous line, it retained its position at the end of the line. MicroEMACS remembers the cursor's position on the line, and returns the cursor there when it jumps to a line long enough to have a character in that position.

Continue pressing **<ctrl-P>**. The cursor will move up the screen until it reaches the top of your text.

Repetitive Motion

Some computers repeat a command automatically if you *hold down* the control key and the character key. Try holding down **<ctrl-N>** for a moment, and see if it repeats automatically. If it does, that will speed moving your cursor around the screen, because you will not have to type the same command repeatedly.

Moving Up and Down by a Screenful of Text

The next two cursor movement commands allow you to roll forward or backwards by one screenful of text.

<ctrl-V>	Move forward one screen
<esc>V	Move back one screen

If you are editing a file with MicroEMACS that is too big to be displayed on your screen all at once, MicroEMACS displays the file in screen-sized portions (on most terminals, 22 lines at a time). The *view* commands **<ctrl-V>** and **<esc>V** allow you to roll up or down one screenful of text at a time.

Type **<ctrl-V>**. Your screen now contains only the last three lines of the file. This is because you have rolled forward by the equivalent of one screenful of text, or 22 lines.

Now, type **<esc>V**. Notice that your text rolls back onto the screen, and your cursor is positioned in the upper left-hand corner of the screen, over the character '/' in the first line.

Moving to Beginning or End of Text

These last two cursor movement commands allow you to jump immediately to the beginning or end of your text.

<esc><	Move to beginning of text
<esc>>	Move to end of text

The *end of text* command **<esc>>** moves the cursor to the end of your text. Type **<esc>>**. Be sure to type a greater-than symbol '>'; this symbol may have been placed anywhere on your keyboard, although on IBM-style keyboards it appears above the period. Your cursor has jumped to the end of your text.

The *beginning of text* command **<esc><** will move the cursor back to the beginning of your text. Type **<esc><**. Be sure to type a less-than symbol '<'; on IBM-style keyboards it appears above the comma. The cursor has jumped back to the upper left-hand corner of your screen.

These commands move you immediately to the beginning or the end of your text, regardless of whether the text is one page or 20 pages long.

Saving Text and Quitting

If you do not wish to continue working at this time, you should *save* your text, and then *quit*.

It is good practice to save your text file every so often while you are working on it. If an accident occurs, such as a power failure, you will not lose all of your work. You can save your text with the *save* command **<ctrl-X><ctrl-S>**. Type **<ctrl-X><ctrl-S>** that is, first type **<ctrl-X>**, then type **<ctrl-S>**. If you had modified this file, the following message would appear:

```
[Wrote 23 lines]
```

The text file would have been saved to your computer's disk. (MicroEMACS sends you messages from time to time. The messages enclosed in square brackets '[' ']' are for your information, and do not necessarily mean that something is wrong.) To exit from MicroEMACS, type the *quit* command **<ctrl-X><ctrl-C>**. This will return you to COHERENT.

Killing and Deleting

Now that you know how to move the cursor, you are ready to edit your text.

To return to MicroEMACS, type the command:

```
me example1.c
```

Within a moment, **example1.c** will be restored to your screen.

By now, you probably have noticed that MicroEMACS is always ready to insert material into your text. Unless you use the **<ctrl>** or **<esc>** keys, MicroEMACS assumes that whatever you type is text and inserts it onto your screen where your cursor is positioned.

The simplest way to erase text is simply to position the cursor to the right of the text you want to erase and backspace over it. MicroEMACS, however, also has a set of commands that allow you to erase text easily. These commands, *kill* and *delete*, behave differently; the distinction is important, and will be explained in a moment.

Deleting Vs. Killing

When MicroEMACS *deletes* text, it is erased completely and disappears forever. However, when MicroEMACS *kills* text, the text is copied into a temporary storage area in memory. This storage area is overwritten when you move the cursor and then kill additional text. Until then, however, the killed text is saved. This aspect of killing allows you to restore text that you killed accidentally, and it also allows you to move or copy portions of text from one position to another.

MicroEMACS is designed so that when it erases text, it does so beginning at the *left edge* of the cursor. This left edge is called the *current position*.

You should imagine that an invisible vertical bar separates the cursor from the character immediately to its left. As you enter the various kill and delete commands, this vertical bar moves to the right or the left with the cursor, and erases the characters it touches.

Erasing Text to the Right

The first two commands to be presented erase text to the *right*.

<ctrl-D>	Delete one character to the right
<esc>D	Kill one word to the right

<ctrl-D> deletes one *character* to the right of the current position. **<esc>D** deletes one *word* to the right of the current position.

To try these commands, type the *delete* command **<ctrl-D>**. MicroEMACS erases the character '/' in the first line, and shifted the rest of the line one space to the left.

Now, type **<esc>D**. MicroEMACS erases the '*' character and the word **This**, and shifts the line six spaces to the left. The cursor is positioned at the *space* before the word **is**. Type **<esc>D** again. The word **is** vanishes along with the *space* that preceded it, and the line shifts *four* spaces to the left.

Remember that **<ctrl-D>** *deletes* text, but **<esc>D** *kills* text.

Erasing Text to the Left

You can erase text to the *left* with the following commands:

<del>	Delete one character to the left
<backspace>	Delete one character to the left
<ctrl-H>	Delete one character to the left
<esc><del>	Kill one word to the left
<esc><backspace>	Kill one word to the left
<esc><ctrl-H>	Kill one word to the left

To see how to erase text to the left, first type the *end of line* command **<ctrl-E>**; this will move the cursor to the right of the word **three** on the first line of text. Now, type **<del>**. The second **e** of the word **three** has vanished.

Type **<esc><del>**. The rest of the word **three** has disappeared, and the cursor has moved to the second space following the word **of**.

Move the cursor four spaces to the left, so that it is over the letter **o** of the word **of**. Type **<esc><del>**. The word **results** has vanished, along with the space that was immediately to the right of it. As before, these commands erased text beginning immediately to the *left* of the cursor. The **<esc><del>** command can be used to erase words throughout your text.

If you wish to erase a word to the left but preserve both spaces that are around it, position the cursor at the space immediately to the right of the word and type **<esc><del>**. If you wish to erase a word to the left plus the space that immediately follows it, position the cursor under the first letter of the *next* word and then type **<esc><del>**.

Typing **<del>** *deletes* text, but typing **<esc><del>** *kills* text.

Erasing Lines of Text

Finally, the following command erases a line of text:

<ctrl-K>	Kill from cursor to end of line

This command kills a line of text, from the line beginning from immediately to the left of the cursor to the end of the line.

To see how this works, move the cursor to the beginning of line 2. Now, strike **<ctrl-K>**. All of line 2 has vanished and been replaced with an empty space. Strike **<ctrl-K>** again. The empty space has vanished, and the cursor is now positioned at the beginning of what used to be line 3, in the space before *** Use**.

Yanking Back (Restoring) Text

The following command allows you restore material that you have killed:

<ctrl-Y>	Yank back (restore) killed text

Remember that when you kill text, MicroEMACS temporarily stores it elsewhere. You can return this material to the screen by using the *yank back* command **<ctrl-Y>**. Type **<ctrl-Y>**. All of line 2 has returned; the cursor, however, remains at the beginning of line 3.

Quitting

When you are finished, do not save the text. If you do so, the undamaged copy of the text that you made earlier will be replaced with the present mangled copy. Rather, use the *quit* command **<ctrl-X><ctrl-C>**. Type **<ctrl-X><ctrl-C>**. On the bottom of your screen, MicroEMACS responds:

```
Quit [y/n]?
```

Reply by typing **y** and a carriage return. If you type **n**, MicroEMACS will return you to where you were in the text. MicroEMACS will now return you to COHERENT.

Block Killing and Moving Text

As noted above, text that is killed is stored temporarily within memory. You can yank killed text back onto your screen, and not necessarily in the spot where it was originally killed. This feature allows you to move text from one position to another.

Moving One Line of Text

You can kill and move one line of text with the following commands:

<ctrl-K>	Kill text to end of line
<ctrl-Y>	Yank back text

To test these commands, invoke MicroEMACS for the file **example1.c** by typing the following command:

```
me example1.c
```

When MicroEMACS appears, the cursor will be positioned in the upper left-hand corner of the screen.

To move the first line of text, begin by typing the *kill* command **<ctrl-K>** twice. Now, press **<esc>>** to move the cursor to the bottom of text. Finally, yank back the line by typing **<ctrl-Y>**. The line that reads

```
/* This is a simple C program that computes the results
```

is now at the bottom of your text.

Your cursor has moved to the point on your screen that is *after* the line you yanked back.

Multiple Copying of Killed Text

When text is yanked back onto your screen, it is *not* deleted from memory. Rather, it is simply copied back onto the screen. This means that killed text can be reinserted into the text more than once. To see how this is done, return to the top of the text by typing **<esc><**. Then type **<ctrl-Y>**. The line you just killed now appears as both the first and last line of the file.

The killed text will not be erased from its temporary storage until you move the cursor and then kill additional text. If you kill several lines or portions of lines in a row, all of the killed text will be stored in the buffer; if you are not careful, you may yank back a jumble of accumulated text.

Kill and Move a Block of Text

If you wish to kill and move more than one line of text at a time, use the following commands:

<ctrl-@>	Set mark
<esc>.	Set mark
<ctrl-W>	Kill block of text
<ctrl-Y>	Yank back text

If you wish to kill a block of text, you can either type the *kill* command **<ctrl-K>** repeatedly to kill the block one line at a time, or you can use the *block kill* command **<ctrl-W>**. To use this command, you must first set a *mark* on the screen, an invisible character that acts as a signal to the computer. The mark can be set with either **<esc>.** or **<ctrl-@>**.

Once the mark is set, you must move your cursor to the other end of the block of text you wish to kill, and then strike **<ctrl-W>**. The block of text will be erased, and will be ready to be yanked back elsewhere.

Try this out on **example1.c**. Type **<esc><** to move the cursor to the upper left-hand corner of the screen. Then type the *set mark* command **<ctrl-@>**. (By the way, be sure to type '@', not '2'.) MicroEMACS will respond with the message

```
[Mark set]
```

at the bottom of your screen. Now, move the cursor down six lines, and type **<ctrl-W>**. Note how the block of text you marked out has disappeared.

Move the cursor to the bottom of your text. Type **<ctrl-Y>**. The killed block of text has now been reinserted.

When you yank back text, be sure to position the cursor at the *exact* point where you want the text to be yanked back. This will ensure that the text will be yanked back in the proper place. To try this out, move your cursor up six lines. Be careful that the cursor is at the *beginning* of the line. Now, type **<ctrl-Y>** again. The text reappeared *above* where the cursor was positioned, and the cursor has not moved from its position at the beginning of

the line which is not what would have happened had you positioned it in the middle or at the end of a line.

Although the text you are working with has only 23 lines, you can move much larger portions of text using only these three commands. Remember, too, that you can use this technique to duplicate large portions of text at several positions to save yourself considerable time in typing and reduce the number of possible typographical errors.

Capitalization and Other Tools

The next commands perform a number of tasks to help with your editing. Before you begin this section, destroy the old text on your screen with the *quit* command **<ctrl-X><ctrl-C>**, and read into MicroEMACS a fresh copy of the program, as you did earlier.

Capitalization and Lowercasing

The following MicroEMACS commands automatically capitalize a word (that is, make the first letter of a word upper case), or make an entire word upper case or lower case.

<esc>C	Capitalize a word
<esc>L	Lowercase an entire word
<esc>U	Uppercase an entire word

To try these commands, do the following: First, move the cursor to the letter **d** of the word *different* on line 2. Type the *capitalize* command **<esc>C**. The word is now capitalized, and the cursor is now positioned at the space after the word. Move the cursor forward so that it is over the letter **t** in **rates**. Press **<esc>C** again. The word changes to **raTes**. When you press **<esc>C**, MicroEMACS capitalizes the *first* letter the cursor meets.

MicroEMACS can also change a word to all upper case or all lower case. (There is very little need for a command that will change only the first character of an upper-case word to lower case, so it is not included.)

Type **<esc>B** to move the cursor so that it is again to the left of the word **Different**. It does not matter if the cursor is directly over the **D** or at the space to its left; as you will see, this means that you can capitalize or lowercase a number of words in a row without having to move the cursor.

Type the *uppercase* command **<esc>U**. The word is now spelled **DIFFERENT**, and the cursor has jumped to the space after the word.

Again, move the cursor to the left of the word **DIFFERENT**. Type the *lowercase* command **<esc>L**. The word has changed back to **different**. Now, move the cursor to the space at the beginning of line 3 by typing **<ctrl-N>** then **<ctrl-A>**. Type **<esc>L** once again. The character "*" is not affected by the command, but the letter **U** is now lower case. **<esc>L** not only shifts a word that is all upper case to lower case: it can also un-capitalize a word.

The *uppercase* and *lowercase* commands stop at the first punctuation mark they meet *after* the first letter they find. This means that, for example, to change the case of a word with an apostrophe in it you must type the appropriate command twice.

Transpose Characters

MicroEMACS allows you to reverse the position of two characters, or *transpose* them, with the *transpose* command **<ctrl-T>**.

Type **<ctrl-T>**. MicroEMACS transposes the character that is under the cursor with the character immediately to its *left*. In this example,

```
* use this
```

in line 3 now appears:

```
* us ethis
```

The space and the letter **e** have been transposed. Type **<ctrl-T>** again. The characters have returned to their original order.

Screen Redraw

<ctrl-L>	Redraw screen

Occasionally, while you are working on a text another COHERENT user will write or mail you a message. COHERENT will write the message directly on your screen, which scrambles your screen. A message sent from another user or a message from the COHERENT system is *not* recorded into your text; however, you may wish to erase the message and continue editing. The *redraw screen* command **<ctrl-L>** will redraw your screen to the way it was before it was scrambled.

Type **<ctrl-L>**. Notice how the screen flickers and the text is rewritten. Had your screen been spoiled by extraneous material, that material would have been erased and the original text rewritten.

The **<ctrl-L>** command also has another use: it can move the line on which the cursor is positioned to the center of the screen. If you have a file that contains more than one screenful of text and you wish to have that particular line in the center of the screen, position the cursor on that line and type **<ctrl-U><ctrl-L>**. Immediately, MicroEMACS redraws the screen, and places the line you selected in the center of the screen.

Return Indent

<ctrl-J>	Return and indent

You may often be faced with a situation in which, for the sake of programming style, you need to indent many lines of text: before every line you must tab the correct number of times before typing the text. These *block indents* can be a time-consuming typing chore. The MicroEMACS **<ctrl-J>** command makes this task easier. **<ctrl-J>** moves the cursor to the next line on the screen and automatically positions the cursor at the previous line's level of indentation.

To see how this works, first move the cursor to the line that reads

```
w3 *= 1.10;
```

Press **<ctrl-E>**, to move the cursor to the end of the line. Now, type **<ctrl-J>**.

As you can see, a new line opens up and the cursor is indented the same amount as the previous line. Type

```
/* Here is an example of auto-indentation */
```

This line of text begins directly under the previous line.

Word Wrap

<ctrl-X>F Set word wrap

Although you have not yet had much opportunity to use it, MicroEMACS will automatically wrap text that you are typing. Word-wrapping is controlled with the *word wrap* command **<ctrl-X>F**. To see how the word wrap command works, first exit from MicroEMACS by typing **<ctrl-X><ctrl-C>**; then reinvoke MicroEMACS by typing

```
me cucumber
```

When MicroEMACS re-appears, type the following text; however, do *not* type any carriage returns:

```
A cucumber should be
well sliced, and dressed
with pepper and vinegar,
and then thrown out, as
good for nothing.
```

When you reached the edge of your screen, a dollar sign was printed and you were allowed to continue typing. MicroEMACS accepted the characters you typed, but it placed them at a location beyond the right edge of your screen.

Now, move to the beginning of the next line and type **<ctrl-U>**. MicroEMACS will reply with the message:

```
Arg: 4
```

Type **30**. The line at the bottom of your screen now appears as follows:

```
Arg: 30
```

(The use of the *argument* command **<ctrl-U>** will be explained in a few minutes.) Now type the *word-wrap* command **<ctrl-X>F**. MicroEMACS will now say at the bottom of your screen:

```
[Wrap at column 30]
```

This sequence of commands has set the word-wrap function, and told it to wrap to the next line all words that extend beyond the 30th column on your screen.

The *word wrap* feature automatically moves your cursor to the beginning of the next line once you type past a preset border on your screen. When you first enter MicroEMACS, that limit is automatically set at the first column, which in effect means that word wrap has been turned off.

When you type prose for a report or a letter of some sort, you probably will want to set the border at the 65th column, so that the printed text will fit neatly onto a sheet of paper. If you are using MicroEMACS to type in a program, however, you probably will want to leave word wrap off, so you do not accidentally introduce carriage returns into your code.

To test word wrapping, type the above text again, without using the carriage return key. When you finish, it should appear as follows:

```
A cucumber should be well
sliced, and dressed with
pepper and vinegar, and then
thrown out, as good for nothing.
```

MicroEMACS automatically moved your cursor to the next line when you typed a space character after the 30th column on your screen.

If you wish to fix the border at some special point on your screen but do not wish to go through the tedium of figuring out how many columns from the left it is, simply position the cursor where you want the border to be, type **<ctrl-X>F**, and then type a carriage return. When **<ctrl-X>F** is typed without being preceded by a **<ctrl-U>** command, it sets the word-wrap border at the point your cursor happens to be positioned. When you do this, MicroEMACS will then print a message at the bottom of your terminal that tells you where the word-wrap border is now set.

To re-word wrap the text between the cursor and the mark, type **<ctrl-X>H**.

If you wish to turn off the word wrap feature again, simply set the word wrap border to one.

Search and Reverse Search

When you edit a large text, you may wish to change particular words or phrases. To do this, you can roll through the text and read each line to find them; or you can have MicroEMACS find them for you. Before you continue, close the present file by typing **<ctrl-X> <ctrl-C>**; then reinvoke the editor to edit the file **example1.c**, as you did before. The following sections perform some exercises with this file.

Search Forward

<ctrl-S>	Search forward incrementally
<esc>S	Search forward with prompt

As you can see from the display, MicroEMACS has two ways to search forward: incrementally, and with a prompt.

An *incremental* search is one in which the search is performed as you type the characters. To see how this works, first type the *beginning of text* command **<esc><** to move the cursor to the upper left-hand corner of your screen. Now, type the *incremental search* command **<ctrl-S>**. MicroEMACS will respond by prompting with the message

```
i-search forward:
```

at the bottom of the screen.

We will now search for the pointer ***msg**. Type the letters ***msg** one at a time, starting with *****. The cursor has jumped to the first place that a * was found: at the second character of the first line. The cursor moves forward in the text file and the message at the bottom of the screen changes to reflect what you have typed.

Now type **m**. The cursor has jumped ahead to the letter **s** in ***msg**. Type **s**. The cursor has jumped ahead to the letter **g** in ***msg**. Finally, type **g**. The cursor is over the space after the token ***msg**. Finally, type **<esc>** to end the string. MicroEMACS replies with the message

```
[Done]
```

which indicates that the search is finished.

If you attempt an incremental search for a word that is not in the file, MicroEMACS will find as many of the letters as it can, and then give you an error message. For example, if you tried to search incrementally for the word ***msgs**, MicroEMACS would move the cursor to the phrase ***msg**; when you typed 's', it would tell you

```
failing i-search forward: *msgs
```

With the *prompt search*, however, you type in the word all at once. To see how this works, type **<esc><**, to return to the top of the file. Now, type the *prompt search* command **<esc>S**. MicroEMACS responds by prompting with the message

```
Search [*msgs]:
```

at the bottom of the screen. The word ***msgs** is shown because that was the last word for which you searched, and so it is kept in the search buffer.

Type in the words **editing text**, then press the carriage return. Notice that the cursor has jumped to the period after the word **text** in the next to last line of your text. MicroEMACS searched for the words **editing text**, found them, and moved the cursor to them.

If the word you were searching for was not in your text, or at least was not in the portion that lies between your cursor and the end of the text, MicroEMACS would not have moved the cursor, and would have displayed the message

```
Not found
```

at the bottom of your screen.

Reverse Search

<ctrl-R>	Search backwards incrementally
<esc>R	Search backwards with prompt

The search commands, useful as they are, can only search forward through your text. To search backwards, use the reverse search commands **<ctrl-R>** and **<esc>R**. These work exactly the same as their forward-searching counterparts, except that they search toward the beginning of the file rather than toward the end.

For example, type **<esc>R**. MicroEMACS replies with the message

```
Reverse search [editing text]:
```

at the bottom of your screen. The words in square brackets are the words you entered earlier for the *search* command; MicroEMACS remembered them. If you wanted to search for **editing text** again, you would just press the carriage return. For now, however, type the word **program** and press the carriage return.

Notice that the cursor has jumped so that it is under the letter **p** of the word **program** in line 1. When you search forward, the cursor moves to the *space after* the word for which you are searching, whereas when you reverse search the cursor moves to the *first letter* of the word for which you are searching.

Cancel a Command

<ctrl-G> Cancel a search command

As you have noticed, the commands to move the cursor or to delete or kill text all execute immediately. Although this speeds your editing, it also means that if you type a command by mistake, it executes before you can stop it.

The *search* and *reverse search* commands, however, wait for you to respond to their prompts before they execute. If you type **<esc>S** or **<esc>R** by accident, MicroEMACS will interrupt your editing and wait for you to initate a search that you do not want to perform. You can evade this problem, however, with the *cancel* command **<ctrl-G>**. This command tells MicroEMACS to ignore the previous command.

To see how this command works, type **<esc>R**. When the prompt appears at the bottom of your screen, type **<ctrl-G>**. Three things happen: your terminal beeps, the characters **^G** appear at the bottom of your screen, and the cursor returns to where it was before you first typed **<esc>R**. The **<esc>R** command has been cancelled, and you are free to continue editing.

If you cancel an *incremental search* command, **<ctrl-S>** or **<esc-S>**, the cursor returns to where it was before you began the search. For example, type **<esc><** to return to the top of the file. Now type **<ctrl-S>** to begin an incremental search, and type **m**. When the cursor moves to the **m** in **simple**, type **<ctrl-G>**. The bell rings, and your cursor returns to the top of the file, where you began the search.

Search and Replace

<esc>% Search and replace

MicroEMACS also gives you a powerful function that allows you to search for a string and replace it with a keystroke. You can do this by executing the *search and replace* command **<esc>%**.

To see how this works, move to the top of the text file by typing **<esc><**; then type **<esc>%**. You will see the following message at the bottom of your screen:

```
Old string:
```

As an exercise, type **msg**. MicroEMACS will then ask:

```
New string:
```

Type **message**, and press the carriage return. As you can see, the cursor jumps to the first occurrence of the string **msg**, and prints the following message at the bottom of your screen:

```
Query replace: [msg] -> [message]
```

MicroEMACS is asking if it should proceed with the replacement. Type a carriage return: this displays the options that are available to you at the bottom of your screen:

```
<SP>[,] replace, [.] rep-end, [n] dont, [!] repl rest <C-G> quit
```

The options are as follows:

Typing a space or a comma executes the replacement, and moves the cursor to the next occurrence of the old string; in this case, it replaces **msg** with **message**, and moves the cursor to the next occurrence of **msg**.

Typing a period '.' replaces this one occurrence of the old string and ends the search and replace procedure. In this example, typing a period replaces this one occurrence of **msg** with **message** and ends the procedure.

Typing the letter 'n' tells MicroEMACS *not* to replace this instance of the old string, but move to the next occurrence of the old string. In this case, typing 'n' does *not* replace **msg** with **message**, and the cursor jumps to the next place where **msg** occurs.

Typing an exclamation point '!' tells MicroEMACS to replace all instances of the old string with the new string automatically, without checking with you any further. In this example, typing '!' replaces all instances of **msg** with **message** without further queries from MicroEMACS.

Finally, typing **<ctrl-G>** aborts the search and replace procedure.

Saving Text and Exiting

This set of basic editing commands allows you to save your text and exit from the MicroEMACS program. They are as follows:

<ctrl-X><ctrl-S>	Save text
<ctrl-X><ctrl-W>	Write text to a new file
<ctrl-Z>	Save text and exit
<ctrl-X><ctrl-C>	Exit without saving text

You have used two of these commands already: the *save* command **<ctrl-X><ctrl-S>** and the *quit* command **<ctrl-X><ctrl-C>**, which respectively allow you to save text or to exit from MicroEMACS without saving text. (Commands that begin with **<ctrl-X>** are called *extended* commands; they are used frequently in the commands described later in this tutorial.)

Write Text to a New File

<ctrl-X> <ctrl-W>	Write text to a new file

If you wish, you can copy the text you are currently editing to a text file other than the one from which you originally read the text. Do this with the *write* command **<ctrl-X><ctrl-W>**.

To test this command, type **<ctrl-X><ctrl-W>**. MicroEMACS displays the following message on the bottom of your screen:

```
Write file:
```

MicroEMACS is asking for the name of the file into which you wish to write the text. Type **sample**. MicroEMACS replies:

```
[Wrote 23 lines]
```

The 23 lines of your text have been copied to a new file called **sample**. The status line at the bottom of your screen has changed to read as follows:

```
-- MicroEMACS -- example1.c -- File: sample -------------
```

The significance of the change in file name will be discussed in the second half of this tutorial.

Before you copy text into a new file, be sure that you have not selected a file name that is already being used. If you do, MicroEMACS will erase whatever is stored under that file name, and the text created with MicroEMACS will be stored in its place.

Save Text and Exit

Finally, the *store* command **<ctrl-Z>** will save your text *and* move you out of the MicroEMACS editor. To see how this works, watch the bottom line of your terminal carefully and type **<ctrl-Z>**. MicroEMACS has saved your text, and now you can issue commands directly to COHERENT.

Advanced Editing

The second half of this tutorial introduces the advanced features of MicroEMACS.

The techniques described here will help you execute complex editing tasks with minimal trouble. You will be able to edit more than one text at a time, display more than one file on your screen at a time, enter a long or complicated phrase repeatedly with only one keystroke, and give commands to COHERENT without having to exit from MicroEMACS.

Before beginning, however, you must prepare a new text file. Type the following command to COHERENT:

```
me example2.c
```

In a moment, **example2.c** will appear on your screen, as follows:

```
/* Use this program to get better acquainted
 * with the MicroEMACS interactive screen editor.
 * You can use this text to learn some of the
 * more advanced editing features of MicroEMACS.
 */

#include <stdio.h>
main()
{
        FILE *fp;
        int ch;
        int filename[20];

        printf("Enter file name: ");
        gets(filename);

        if ((fp =fopen(filename,"r")) !=NULL) {
                while ((ch = fgetc(fp)) != EOF)
                        fputc(ch, stdout);
        } else
                printf("Cannot open %s.\n", filename);
        fclose(fp);
}
```

Arguments

Most of the commands already described in this tutorial can be used with *arguments*. An argument is a subcommand that tells MicroEMACS to execute a command a given number of times. With MicroEMACS, arguments are introduced by typing **<ctrl-U>**.

Arguments: Default Values

By itself, **<ctrl-U>** sets the argument at *four*. To illustrate this, first type the *next line* command **<ctrl-N>**. By itself, this command moves the cursor down one line, from being over the '/' at the beginning of line 1, to being over the *space* at the beginning of line 2.

Now, type **<ctrl-U>**. MicroEMACS replies with the message:

```
Arg: 4
```

Now type **<ctrl-N>**. The cursor jumps down *four* lines, from the beginning of line 2 to the letter **m** of the word **main** at the beginning of line 6.

Type **<ctrl-U>**. The line at the bottom of the screen again shows that the value of the argument is four. Type **<ctrl-U>** again. Now the line at the bottom of the screen reads:

```
Arg: 16
```

Type **<ctrl-U>** once more. The line at the bottom of the screen now reads:

```
Arg: 64
```

Each time you type **<ctrl-U>**, the value of the argument is *multiplied* by four. Type the *forward* command **<ctrl-F>**. The cursor has jumped ahead 64 characters, and is now over the **i** of the word **file** in the *printf* statement in line 11.

Selecting Values

Naturally, an argument does not have to be a power of four. You can set the argument to whatever number you wish, simply by typing **<ctrl-U>** and then typing the number you want.

For example, type **<ctrl-U>**, and then type **3**. The line at the bottom of the screen now reads:

```
Arg: 3
```

Type the *delete* command **<esc>D**. MicroEMACS has deleted three words to the right.

You can use arguments to increase the power of any *cursor movement* command, or any *kill* or *delete* command. The sole exception is **<ctrl-W>**, the *block kill* command.

Deleting With Arguments: An Exception

Killing and *deleting* were described in the first part of this tutorial. They were said to differ in that text that was killed was stored in a special area of the computer and could be yanked back, whereas text that was deleted was erased outright. However, there is one exception to this rule: any text that is deleted using an argument can also be yanked back.

To see how this works, first type the *begin text* command **<esc><** to move the cursor to the upper left-hand corner of the screen. Then, type **<ctrl-U> 5 <ctrl-D>**. The word **Use** has disappeared. Move the cursor to the right until it is between the words **better** and **acquainted**, then type **<ctrl-Y>**. The word **Use** has been moved within the line (although the spaces around it have not been moved). This function is very handy, and should greatly speed your editing.

Remember, too, that unless you move the cursor between one set of deletions and another, the computer's storage area will not be erased, and you may yank back a jumble of text.

Buffers and Files

Before beginning this section, replace the edited copy of the text on your screen with a fresh copy. Type the *quit* command **<ctrl-X><ctrl-C>** to exit from MicroEMACS without saving the text; then return to MicroEMACS to edit the file **example2.c**, as you did earlier.

Now, look at the status line at the bottom of your screen. It should appear as follows: As noted in the first half of this tutorial, the name on the left of the command line is that of the program. The name in the middle is the name of the *buffer* with which you are now working, and the name to the right is the name of the *file* from which you read the text.

Definitions

A *file* is a mass of text that has been given a name and has been permanently stored on your disk. A *buffer* is a portion of the computer's memory that has been set aside for you to use, which may be given a name, and into which you can put text temporarily. You can place text into the buffer either by typing it at your keyboard or by *copying* it from a file.

Unlike a file, a buffer is not permanent: if your computer were to stop working (because you turned the power off, for example), a file would not be affected, but a buffer would be erased.

You must *name* your files because you work with many different files, and you must have some way to tell them apart. Likewise, MicroEMACS allows you to *name* your buffers, because MicroEMACS allows you to work with more than one buffer at a time.

File and Buffer Commands

MicroEMACS gives you a number of commands for handling files and buffers. These include the following:

<ctrl-X><ctrl-W>	Write text to file
<ctrl-X><ctrl-F>	Rename file
<ctrl-X><ctrl-R>	Replace buffer with named file
<ctrl-X><ctrl-V>	Switch buffer or create a new buffer
<ctrl-X>K	Delete a buffer
<ctrl-X><ctrl-B>	Display the status of each buffer

Write and Rename Commands

The *write* command **<ctrl-X><ctrl-W>** was introduced earlier when the commands for saving text and exiting were discussed. To review, **<ctrl-X><ctrl-W>** changes the name of the file into which the text is saved, and then copies the text into that file.

Type **<ctrl-X><ctrl-W>**. MicroEMACS responds by printing

```
Write file:
```

on the last line of your screen.

Type **junkfile**, then **<return>**. Two things happen: First, MicroEMACS writes the message

```
[Wrote 21 lines]
```

at the bottom of your screen. Second, the name of the file shown on the status line changes from **example2.c** to **junkfile**. MicroEMACS is reminding you that your text is now being saved into the file **junkfile**.

The *file rename* command **<ctrl-X><ctrl-F>** allows you rename the file to which you are saving text, *without* automatically writing the text to it. Type **<ctrl-X><ctrl-F>**. MicroEMACS will reply with the prompt:

```
Name:
```

Type **example2.c** and **<return>**. MicroEMACS does *not* send you a message that lines were written to the file; however, the name of the file shown on the status line has changed from **junkfile** back to **example2.c**.

Replace Text in a Buffer

The *replace* command **<ctrl-X><ctrl-R>** allows you to replace the text in your buffer with the text from another file.

Suppose, for example, that you had edited **example2.c** and saved it, and now wished to edit **example1.c**. You could exit from MicroEMACS, then re-invoke MicroEMACS for the file **example2.c**, but this is cumbersome. A more efficient way is to simply replace the **example2.c** in your buffer with **example1.c**.

Type **<ctrl-X><ctrl-R>**. MicroEMACS replies with the prompt:

```
Read file:
```

Type **example1.c**. Notice that **example2.c** has rolled away and been replaced with **example1.c**. Now, check the status line. Notice that although the name of the *buffer* is still **example2.c**, the name of the *file* has changed to **example1.c**. You can now edit **example1.c**; when you save the edited text, MicroEMACS will copy it back into the file **example1.c** unless, of course, you again choose to rename the file.

Visiting Another Buffer

The last command of this set, the *visit* command **<ctrl-X><ctrl-V>**, allows you to create more than one buffer at a time, to jump from one buffer to another, and move text between buffers. This powerful command has numerous features.

Before beginning, however, straighten up your buffer by replacing **example1.c** with **example2.c**. Type the *replace* command **<ctrl-X><ctrl-R>**; when MicroEMACS replies by asking

```
Read file:
```

at the bottom of your screen, type **example2.c**.

You should now have the file **example2.c** read into the buffer named **example2.c**.

Now, type the *visit* command **<ctrl-X><ctrl-V>**. MicroEMACS replies with the prompt

```
Visit file:
```

at the bottom of the screen. Now type **example1.c**. Several things happen. **example2.c** rolls off the screen and is replaced with **example1.c**; the status line changes to show that both the buffer name and the file name are now **example1.c**; and the message

```
[Read 23 lines]
```

appears at the bottom of the screen.

This does *not* mean that your previous buffer has been erased, as it would have been had you used the *replace* command **<ctrl-X><ctrl-R>**. MicroEMACS is still keeping **example2.c** "alive" in a buffer and it is available for editing; however, it is not being shown on your screen at the present moment.

Type **<ctrl-X><ctrl-V>** again, and when the prompt appears, type **example2.c**. **example1.c** scrolls off your screen and is replaced by **example2.c**, and the message

```
[Old buffer]
```

appears at the bottom of your screen. You have just jumped from one buffer to another.

Move Text From One Buffer to Another

The *visit* command **<ctrl-X><ctrl-V>** not only allows you to jump from one buffer to another: it allows you to *move text* from one buffer to another as well. The following example shows how you can do this.

First, kill the first line of **example2.c** by typing the *kill* command **<ctrl-K>** twice. This removes both the line of text *and* the space that it occupied. If you did not remove the space as well as the line itself, no new line would be created for the text when you yank it back. Next, type **<ctrl-X><ctrl-V>**. When the prompt

```
Visit file:
```

appears at the bottom of your screen, type **example1.c**. When **example1.c** has rolled onto your screen, type the *yank back* command **<ctrl-Y>**. The line you killed in **example2.c** has now been moved into **example1.c**.

Checking Buffer Status

The number of buffers you can use at any one time is limited only by the size of your computer. You should create only as many buffers as you need to use immediately; this will help the computer run efficiently.

To help you keep track of your buffers, MicroEMACS has the *buffer status* command **<ctrl-X><ctrl-B>**. Type **<ctrl-X><ctrl-B>**. The status line moves up to the middle of the screen, and the bottom half of your screen is replaced with the following display:

```
C    Size Lines   Buffer         File
-    ---- -----   ------         ----
*     655    24   example1.c     example1.c
*     403    20   example2.c     example2.c
```

This display is called the *buffer status window*. The use of windows will be discussed more fully in the following section.

The letter **C** over the leftmost column stands for **Changed**. An asterisk indicates that that buffer has been changed since it was last saved, whereas a space means that the buffer has not been changed. **Size** indicates the buffer's size, in number of characters; **Buffer** lists the buffer name, and **File** lists the file name.

Now, kill the second line of **example1.c** by typing the *kill* command **<ctrl-K>**. Then type **<ctrl-X><ctrl-B>** once again. The size of the buffer **example1.c** shrinks from 657 characters to 595 to reflect the decrease in the size of the buffer.

To make this display disappear, type the *one window* command **<ctrl-X>1**. This command will be discussed in full in the next section.

Renaming a Buffer

One more point must be covered with the *visit* command. COHERENT does not allow you to have more than one file with the same name. For the same reason, MicroEMACS does not allow you to have more than one *buffer* with the same name.

Ordinarily, when you visit a file that is not already in a buffer, MicroEMACS creates a new buffer and gives it the same name as the file you are visiting. However, if for some reason you already have a buffer with the same name as the file you wish to visit, MicroEMACS stops and asks you to give a new, different name to the buffer it is creating.

For example, suppose that you wanted to visit a new *file* named **sample**, but you already had a *buffer* named **sample**. MicroEMACS would stop and give you this prompt at the bottom of the screen:

```
Buffer name:
```

You would type in a name for this new buffer. This name could not duplicate the name of any existing buffer. MicroEMACS would then read the file **sample** into the newly named buffer.

Delete a Buffer

If you wish to delete a buffer, simply type the *delete buffer* command **<ctrl-X>K**. This command allows you to delete only a buffer that is hidden, not one that is being displayed.

Type **<ctrl-X>K**. MicroEMACS will give you the prompt:

```
Kill buffer:
```

Type **example2.c**. Because you have changed the buffer, MicroEMACS asks:

```
Discard changes [y/n]?
```

Type **y**. Now, type the *buffer status* command **<ctrl-X><ctrl-B>**. The buffer status window no longer shows the buffer **example2.c**. Although the prompt refers to *killing* a buffer, the buffer is in fact *deleted* and cannot be yanked back.

Windows

Before beginning this section, it will be necessary to create a new text file. Exit from MicroEMACS by typing the *quit* command **<ctrl-X><ctrl-C>**; then reinvoke MicroEMACS for the text file **example1.c** as you did earlier.

Now, copy **example2.c** into a buffer by typing the **visit** command **<ctrl-X><ctrl-V>**. When the message

```
Visit file:
```

appears at the bottom of your screen, type **example2.c**. MicroEMACS reads **example2.c** into a buffer, and shows the message

```
[Read 21 lines]
```

at the bottom of your screen.

Finally, copy a new text, called **example3.c**, into a buffer. (You can find it in the same place where the files **example1.c** and **example2.c** are kept.) Type **<ctrl-X><ctrl-V>** again. When MicroEMACS asks which file to visit, type **example3.c**. The message

```
[Read 123 lines]
```

appears at the bottom of your screen.

The first screenful of text appears as follows:

```
/*
 * Factor prints out the prime factorization of numbers.
 * If there are any arguments, then it factors these.  If
 * there are no arguments, then it reads stdin until
 * either EOF or the number zero or a non-numeric
 * non-white-space character.  Since factor does all of
 * its calculations in double format, the largest number
 * which can be handled is quite large.
 */
#include <stdio.h>
#include <math.h>
#include <ctype.h>
```

```
#define NUL '\0'
#define ERROR 0x10 /* largest input base */
#define MAXNUM 200 /* max number of chars in number */

main(argc, argv)
int argc;
register char *argv[];

-- MicroEMACS -- example3.c -- File: example3.c --------------
```

At this point, **example3.c** is on your screen, and **example1.c** and **example2.c** are hidden.

You could edit first one text and then another, while remembering just how things stood with the texts that were hidden; but it would be much easier if you could display all three texts on your screen simultaneously. MicroEMACS allows you to do just that by using *windows*.

Creating Windows and Moving Between Them

A *window* is a portion of your screen that can be manipulated independent of the rest of the screen. The following commands let you create windows and move between them:

<ctrl-X>2	Create a window
<ctrl-X>1	Delete extra windows
<ctrl-X>N	Move to next window
<ctrl-X>P	Move to previous window

The best way to grasp how a window works is to create one and work with it. To begin, type the *create a window* command **<ctrl-X>2**.

Your screen is now divided into two parts, an upper and a lower. The same text is in each part, and the command lines give **example3.c** for the buffer and file names. Also, note that you still have only one cursor, which is in the upper left-hand corner of the screen.

The next step is to move from one window to another. Type the *next window* command **<ctrl-X>N**. Your cursor has now jumped to the upper left-hand corner of the *lower* window.

Type the *previous window* command **<ctrl-X>P**. Your cursor has returned to the upper left-hand corner of the top window.

Now, type **<ctrl-X>2** again. The window on the top of your screen is now divided into two windows, for a total of three on your screen. Type **<ctrl-X>2** again. The window at the top of your screen has again divided into two windows, for a total of four.

It is possible to have as many as 11 windows on your screen at one time, although each window will show only the control line and one or two lines of text. Neither **<ctrl-X>2** nor **<ctrl-X>1** can be used with arguments.

Now, type the *one window* command **<ctrl-X>1**. All of the extra windows have been eliminated, or *closed*.

Enlarging and Shrinking Windows

When MicroEMACS creates a window, it divides into half the window in which the cursor is positioned. You do not have to leave the windows at the size MicroEMACS creates them, however. If you wish, you may adjust the relative size of each window on your screen, using the *enlarge window* and *shrink window* commands:

<ctrl-X>Z	Enlarge window
<ctrl-X><ctrl-Z>	Shrink window

To see how these work, first type **<ctrl-X>2** twice. Your screen is now divided into three windows: two in the top half of your screen, and the third in the bottom half.

Now, type the *enlarge window* command **<ctrl-X>Z**. The window at the top of your screen is now one line bigger: it has borrowed a line from the window below it. Type **<ctrl-X>Z** again. Once again, the top window has borrowed a line from the middle window.

Now, type the *next window* command **<ctrl-X>N** to move your cursor into the middle window. Again, type the *enlarge window* command **<ctrl-X>Z**. The middle window has borrowed a line from the bottom window, and is now one line larger.

The *enlarge window* command **<ctrl-X>Z** allows you to enlarge the window your cursor is in by borrowing lines from another window, provided that you do not shrink that other window out of existence. Every window must have at least two lines in it: one command line and one line of text.

The *shrink window* command **<ctrl-X><ctrl-Z>** allows you to decrease the size of a window. Type **<ctrl-X><ctrl-Z>**. The present window is now one line smaller, and the lower window is one line larger because the line borrowed earlier has been returned.

The *enlarge window* and *shrink window* commands can also be used with arguments introduced with **<ctrl-U>**. However, remember that MicroEMACS will not accept an argument that would shrink another window out of existence.

Displaying Text Within a Window

Displaying text within the limited area of a window can present special problems. The *view* commands **<ctrl-V>** and **<esc>V** roll window-sized portions of text up or down, but you may become disoriented when a window shows only four or five lines of text at a time. Therefore, three special commands are available for displaying text within a window:

<ctrl-X><ctrl-N>	Scroll down
<ctrl-X><ctrl-P>	Scroll up
<esc>!	Move within window

Two commands allow you to move your text by one line at a time, or *scroll* it: the *scroll up* command **<ctrl-X><ctrl-N>**, and the *scroll down* command **<ctrl-X><ctrl-P>**.

Type **<ctrl-X><ctrl-N>**. The line at the top of your window has vanished, a new line has appeared at the bottom of your window, and the cursor is now at the beginning of what had been the second line of your window.

Now type **<ctrl-X><ctrl-P>**. The line at the top that had vanished earlier has now returned, the cursor is at the beginning of it, and the line at the bottom of the window has vanished. These commands allow you to move forward in your text slowly so that you do not become disoriented.

Both of these commands can be used with arguments introduced by **<ctrl-U>**.

The third special movement command is the *move within window* command **<esc>!**. This command moves the line your cursor is on to the top of the window.

To try this out, move the cursor down three lines by typing **<ctrl-U>3<ctrl-N>**; now type **<esc>!**. (Be sure to type an exclamation point '!', not a numeral one '1', or nothing will happen.) The line to which you had moved the cursor is now the first line in the window, and three new lines have scrolled up from the bottom of the window. You will find this command to be very useful as you become more experienced at using windows.

All three special movement commands can also be used when your screen has no extra windows, although you will not need them as much.

One Buffer

Now that you have been introduced to the commands for manipulating windows, you can begin to use windows to speed your editing.

To begin with, scroll up the window you are in until you reach the top line of your text. You can do this either by typing the *scroll up* command **<ctrl-X><ctrl-P>** several times, or by typing **<esc><**.

Kill the first line of text with the *kill* command **<ctrl-K>**. The first line of text has vanished from all three windows. Now, type **<ctrl-Y>** to yank back the text you just killed. The line has reappeared in all three windows.

The main advantage to displaying one buffer with more than one window is that each window can display a different portion of the text. This can be quite helpful if you are editing or moving a large text.

To demonstrate this, do the following: First, move to the end of the text in your present window by typing the *end of text* command **<esc>>**, then typing the *previous line* command **<ctrl-P>** four times. Now, kill the last four lines.

You could move the killed lines to the beginning of your text by typing the *beginning of text* command **<esc><**; however, it is more convenient simply to type the *next window* command **<ctrl-X>N**, which moves you to the beginning of the text as displayed in the next window. MicroEMACS remembers a different cursor position for each window.

Now yank back the four killed lines by typing **<ctrl-Y>**. You can simultaneously observe that the lines have been removed from the end of your text and that they have been restored at the beginning.

Multiple Buffers

Windows are especially helpful when they display more than one text. Remember that at present you are working with *three* buffers, named **example1.c**, **example2.c**, and **example3.c**, although your screen is displaying only the text **example3.c**. To display a different text in a window, use the *switch buffer* command **<ctrl-X>B**.

Type **<ctrl-X>B**. When MicroEMACS asks

```
Use buffer:
```

at the bottom of the screen, type **example1.c**. The text in your present window is replaced with **example1.c**. The command line in that window changes, too, to reflect the fact that the buffer and the file names are now **example1.c**.

Moving and Copying Text Among Buffers

It is now very easy to copy text among buffers. To see how this is done, first kill the first line of **example1.c** by typing the **<ctrl-K>** command twice. Yank back the line immediately by typing **<ctrl-Y>**. Remember, the line you killed has *not* been erased from its special storage area, and may be yanked back any number of times.

Now, move to the previous window by typing **<ctrl-X>P**, then yank back the killed line by typing **<ctrl-Y>**. This technique can also be used with the *block kill* command **<ctrl-W>** to move large amounts of text from one buffer to another.

Checking Buffer Status

The *buffer status command* **<ctrl-X><ctrl-B>** can be used when you are already displaying more than one window on your screen.

When you want to remove the buffer status window, use either the *one window* command **<ctrl-X>1**, or move your cursor into the buffer status window using the *next window* command **<ctrl-X>N** and replace it with another buffer by typing the *switch buffer* command **<ctrl-X>B**.

Saving Text From Windows

The final step is to save the text from your windows and buffers. Close the lower two windows with the *one window* command **<ctrl-X>1**. Remember, when you close a window, the text that it displayed is still kept in a buffer that is *hidden* from your screen. For now, do *not* save any of these altered texts.

When you use the *save* command **<ctrl-X><ctrl-S>**, only the text in the window in which the cursor is positioned is written to its file. If only one window is displayed on the screen, the *save* command will save only its text.

If you made changes to the text in another buffer, such as moving portions of it to another buffer, MicroEMACS would ask

```
Quit [y/n]:
```

If you answer '**n**', MicroEMACS will *save* the contents of the buffer you are currently displaying by writing them to your disk, but it will ignore the contents of other buffers, and your cursor will be returned to its previous position in the text. If you answer '**y**', MicroEMACS again will save the contents of the current buffer and ignore the other buffers, but you will exit from MicroEMACS and return to Exit from MicroEMACS by typing the *quit* command **<ctrl-X><ctrl-C>**.

Keyboard Macros

A *keyboard macro* is a set of MicroEMACS commands that are stored in memory and given a name. After you create a keyboard macro, you can execute it again and again just by typing its name. If you are revising a big file, you will find that keyboard macros are one of the most useful features in MicroEMACS, and one that you will use often.

The following table summarizes MicroEMACS's keyboard-macro commands:

<ctrl-X>(	Open a keyboard macro
<ctrl-X>)	Close a keyboard macro
<ctrl-X>E	Execute a keyboard macro
<ctrl-X>M	Rename a keyboard macro
<ctrl-X>I	Bind current macro as initialization macro

Creating a Keyboard Macro

To begin to create a macro, type the *begin macro* command **<ctrl-X>(**. Be sure to type an open parenthesis '(', not a numeral '9'. MicroEMACS will reply with the message

```
[Start macro]
```

Type the following phrase:

```
MAXNUM
```

Then type the *end macro* command **<ctrl-X>)**. Be sure you type a close parenthesis ')', not a numeral '0'. MicroEMACS will reply with the message

```
[End macro]
```

Move your cursor down two lines and execute the macro by typing the *execute macro* command **<ctrl-X>E**. The phrase you typed into the macro has been inserted into your text.

If you give these commands in the wrong order, MicroEMACS warns you that you are making a mistake. For example, if you open a keyboard macro by typing **<ctrl-X>(**, and then attempt to open another keyboard macro by again typing **<ctrl-X>(**, MicroEMACS will say:

```
Not now
```

Should you accidentally open a keyboard macro, or enter the wrong commands into it, you can cancel the entire macro simply by typing **<ctrl-G>**.

Execute a Macro Repeatedly

As with most MicroEMACS commands, you can use a keyboard macro with an argument to execute it repeatedly. For example, if you have defined a keyboard macro, then typing

<ctrl-U><ctrl-X>E

executes that macro four times. (Remember, four is the default value for **<ctrl-U>**.)

As described above, **<ctrl-U>** normally is used with a positive number, to indicate how often MicroEMACS should execute a given command or macro. With keyboard macros, however, you can use a special value for **<ctrl-U>**: -1. This tells MicroEMACS to repeatedly execute a keyboard macro until it fails.

For example, consider that you define the following keyboard macro:

<ctrl-S> foo **<ctrl-K>**

This macro searches for the string "foo", then kills the rest of line that that string is on. Now, when you type the command

<ctrl-U> -1 <ctrl-X>E

executes this macro until MicroEMACS can no longer find the string "foo"; it then quits.

Obviously, you should define your macro carefully before you execute it with this -1 option to **<ctrl-U>**; otherwise, you can commit tremendous mayhem on your file with one keystroke.

Replacing a Macro

To replace this macro with another, go through the same process. Type **<ctrl-X>(**. Then type the *buffer status* command **<ctrl-X><ctrl-B>**, and type **<ctrl-X>)**. Remove the buffer status window by typing the *one window* command **<ctrl-X>1**.

Now execute your keyboard macro by typing the **execute macro** command **<ctrl-X>E**. The *buffer status* command has executed once more.

Renaming a Macro

Many times during a long editing session, you will find that you use one keyboard macro, then use a second keyboard macro, then find that you need the first macro again. In previous releases of MicroEMACS, the only way to do this work was to type the first macro, replace it with the second macro, then retype the first macro when you need it again. The present edition of MicroEMACS, however, lets you define any number of keyboard macros, and save them by giving each one a unique "name" that is, its own unique keyboard binding.

To rename a keyboard macro that you have already created, use the *rename macro* command **<ctrl-X>M**. To see how this works, do the following: (1) Type **<ctl-X>(** to open the keyboard macro. (2) Now, type **<esc>**s xyz **<ctrl-U> <ctrl-D>** to fill the macro with something. (3) Finally, type **<ctrl-X>)** to close the macro.

Now, type **<ctrl-X>M**, to rename the macro. MicroEMACS will reply with the prompt:

```
enter keybinding for macro
```

Type **<esc>L**. This tells MicroEMACS to take the keyboard macro you created and link it to the keystrokes **<esc>L**.

Now, whenever you type **<esc>L**, MicroEMACS will execute **<esc>**s xyz **<ctrl-U> <ctrl-D>**. You can now define another keyboard macro without wiping out the one you have renamed. There is no theoretical limit to the number of keyboard macros you can create, although there are practical limits imposed by the amount of memory available to MicroEMACS.

Renaming Macros: A Few Caveats

Please note that if you name a keyboard macro with a keystroke that is already defined, MicroEMACS will no longer be able to access that keystroke's functionality.

For example, if instead of naming your new macro **<esc>L** you named it **<ctrl-A>**, then every time you typed **<ctrl-A>** MicroEMACS would execute **<esc>S** xyz **<ctrl-U> <ctrl-D>** and you would no longer be able to jump to the beginning of a line (which **<ctrl-A>** normally does).

The only exceptions are **<ctrl-X>**, **<esc>**, and the **<ctrl-X>R** command (described below), which MicroEMACS will not let you reassign. Obviously, you should be very careful when you assign a name to a keyboard macro, or you could easily clobber much of the editor's functionality.

Note, too, that MicroEMACS lets you define reflexive keybindings, but these never work. For example, if you named the above example macro **<ctrl-D>** instead of **<esc>L**, then every time you typed **<ctrl-D>** MicroEMACS would try to execute a macro that included **<ctrl-D>** in it. Obviously, this can tie MicroEMACS into knots in no time. Again, please be very careful when you assign names to keyboard macros.

The commands **<ctrl-X>S** and **<ctrl-X>L** let you save all named keyboard macros into a file, and restore them during a later editing session. These commands are described in the next section.

Setting the Initialization Macro

MicroEMACS allows one macro to be specified which will be executed every time MicroEMACS is invoked. This "initialization macro" can be set using the key sequence **<ctrl-X>I** and causes MicroEMACS to "bind" the currently defined macro to the initialization macro.

Flexible Key Bindings

As you have noticed by now, MicroEMACS works through standard *key bindings:* that is, one keystroke or combination of keystrokes tells MicroEMACS to perform a particular task. For example, typing **<ctrl-A>** tells MicroEMACS to move the cursor to the beginning of the line; typing **<ctrl-E>** tells MicroEMACS to move the cursor to the end of the line; and so on.

MicroEMACS allows you to change its key bindings, so you can bind a given keystroke or combination of keystrokes to a task other than the default one documented in this tutorial. In this way, you can reconfigure MicroEMACS so that it resembles another editor with which you are more familiar.

To perform this magic, MicroEMACS uses two tables for keybindings: a *default table* that is loaded at compile time and never changes, and a *dynamic table* that you can modify with MicroEMACS's keybinding commands.

The following table summarizes MicroEMACS's commands for flexible keybindings:

<ctrl-X>R	Replace one binding with another
<ctrl-X>X	Rebind prefix keys
<ctrl-X>S	Save flexible bindings and macros into file
<ctrl-X>L	Load flexible bindings and macros from file

Changing a Keybinding

The *replace binding* command **<ctrl-X>R** replaces one binding with another. For example, if you wished to replace the *beginning of line* command **<ctrl-A>** with **<esc>Z**, you would do the following:

1. Type **<ctrl-X>R** to invoke the rebinding command.
2. When you see the prompt

   ```
   Enter old keybinding
   ```

   type the keybinding you wish to change in this case, **<ctrl-A>**.

3 When you then see the prompt

```
Enter new keybinding
```

type the keybinding to which you wish to change it in this case, **<esc>Z**.

Note that you cannot rebind the command **<ctrl-X>R** itself; otherwise, you would paint yourself into a corner. Also, note that if you rebind a command to itself (that is, if you type the same keybinding in response to both prompts), then that keybinding is bound to the old meaning of the keybindings, should there be any.

Rebinding Metakeys

MicroEMACS's keybindings depend on several pre-defined metakeys. A *metakey* is a keystroke that introduces a further set of commands. MicroEMACS's default keybindings use two metakeys: **<ctrl-X>** and **<esc>**. Other editors use other keystrokes as metakeys. If you wish to rebind a metakey, use the *rebind metakey* command **<ctrl-X>X**. This command prompts you to bind up to three metakeys, and the argument key **<ctrl-U>**.

For example, suppose that you wish to change the metakey **<ctrl-X>** to **<ctrl-G>**. To begin, type the command **<ctrl-X>X**. You will see the prompt

```
Enter prefix character 1 or space
```

"Prefix character 1" is **<ctrl-X>** in the default bindings. Type **<ctrl-G>**. You will then see the prompt:

```
Enter prefix character 2 or space
```

"Prefix character 2" is **<esc>** in the default bindings. Since you do not want to change it, type **<space>**. You will then see the prompt:

```
Enter prefix character 3 or space
```

There is no "prefix character 3" in the default bindings, but you can set a third one for your keybindings if you wish. Since (for the sake of this example) you do not wish to set one, type **<space>**. Finally, you will see the prompt:

```
Enter repeat code or space
```

The "repeat code" executes a command repeatedly; in this tutorial, it is often called the "argument key" or "argument command". Since (in this example) you do not wish to change it, type **<space>**.

Now that you have reset the **<ctrl-X>** metakey, you must now type **<ctrl-G>** every time in place of **<ctrl-X>** throughout all of the MicroEMACS commands. For example, if you wished to change the metakey back from **<ctrl-G>** to **<ctrl-X>**, you would have to type **<ctrl-G>X** to invoke the *rebind metakey* command.

Note that because **<ctrl-G>** already is bound in the MicroEMACS keybindings, when you rebind it the command to which it was bound is no longer available to you. However, if you un-rebind the key, then it automatically is rebound to its old command. In the above example, **<ctrl-G>** is bound to the *insert literal character* command, which lets you insert control characters into your file. When you rebound the **<ctrl-X>** metakey to **<ctrl-G>**, then the *insert literal character* command was no longer available to you. However, when you re-rebound the **<ctrl-G>** metakey to **<ctrl-X>**, then **<ctrl-G>** was automatically rebound to the *insert literal character* command.

Save and Restore Keybindings

MicroEMACS lets you save your rebound keybindings into a file, and reload them during another editing session. To save your keybindings into a file, type the *save keybindings* command **<ctrl-X>S**. Try it. You will see the prompt:

```
Store bindings file:
```

Type the name of a file. MicroEMACS then writes its keybindings into that file. Note that this command also saves all named keyboard macros that you have created.

To restore a set of keybindings, use the *restore keybindings* command **<ctrl-X>L**. Try it. You will see the prompt:

```
Load bindings file:
```

Type the name of the find in which you saved the system's keybindings and all named keyboard macros. MicroEMACS will then load them into memory for you.

These commands let you prepare several sets of customized keybindings and macros. You can customize keybindings to suit your preference, or create custom sets of macros to suit any number of specialized editing tasks.

By default, MicroEMACS checks for the existence of file **$HOME/.emacs.rc** and executes it if found. The **-f** option lets you specify an alternate file of keybindings macros from the **me** command line. After loading the file, MicroEMACS then executes the initialization macro, if one exists. For example, if you wish to use the set of keybindings saved in file **keybind** to edit file **textfile**, then you would type the following:

```
me -f keybind textfile
```

As you can see, MicroEMACS's system of keyboard macros and flexible key bindings help make it an extremely flexible and powerful editor.

Sending Commands to COHERENT

The only remaining commands you need to learn are the *program interrupt* commands **<ctrl-X>! and <ctrl-C>**. These commands allow you to interrupt your editing, give a command directly to COHERENT, and then resume editing without affecting your text in any way.

The command **<ctrl-X>!** allows you to send *one* command line (one command, or several commands plus separators) to the operating system. To see how this command works, type **<ctrl>!**. The prompt **!** has appeared at the bottom of your screen. Type **lc**. Observe that the directory's table of contents scrolls across your screen, followed by the message **[end]**. To return to your editing, simply type a carriage return. The *interrupt* command **<ctrl-C>** suspends editing indefinitely, and allows you to send an unlimited number of commands to the operating system. To see how this works, type **<ctrl-C>**. After a moment, the COHERENT system's prompt will appear at the bottom of your screen. Type **time**. The COHERENT system replies by printing the time and date. To resume editing, then simply type **<ctrl-D>**.

If you wish, you can suspend MicroEMACS's operation, tell the COHERENT system to invoke another copy of the MicroEMACS program, edit a file, then return to your previous editing. To see how this is done, type **<ctrl-C>**. When the prompt appears at the bottom of your screen, type

```
me example1.c
```

It doesn't matter that you are already editing **example1.c**. MicroEMACS will simply copy the **example1.c** file into a new buffer and let you work as if the other MicroEMACS program you just interrupted never existed.

Exit from this second MicroEMACS program by typing the *quit* command **<ctrl-X><ctrl-C>**. Then type **<ctrl-D>**. Your original MicroEMACS program has now been resumed. However, none of the changes you made in the secondary MicroEMACS program will be seen here.

It is not a good idea to use multiple MicroEMACS programs to edit the same program: it is too easy to become confused as to which edits were made to which version.

The only time this is advisable is if you wish to test to see how a certain edit would affect your text: you can create a new MicroEMACS program, test the command, and then destroy the altered buffer and return to your original editing program without having to worry that you might make errors that are difficult to correct.

Now type **<ctrl-X><ctrl-C>** to exit.

Compiling and Debugging Through MicroEMACS

MicroEMACS can be used with the compilation command **cc** to give you a reliable system for debugging new programs.

Often, when you're writing a new program, you face the situation in which you try to compile, but the compiler produces error messages and aborts the compilation. You must then invoke your editor, change the program, close the editor, and try the compilation over again. This cycle of compilation editing recompilation can be quite bothersome.

To remove some of the drudgery from compiling, the **cc** command has the *automatic*, or MicroEMACS option, **-A**. When you compile with this option, the MicroEMACS screen editor will be invoked automatically if any errors occur. The error or errors generated during compilation will be displayed in one window, and your text in the other, with the cursor set at the number of the line that the compiler indicated had the error.

Try the following example. Use MicroEMACS to enter the following program, which you should call **error.c**:

```
main() {
        printf("Hello, world!\n")
}
```

The semicolon was left off of the **printf** statement, which is an error. Now, try compiling **error.c** with the following **cc** command:

```
cc -A error.c
```

You should see no messages from the compiler because they are all being diverted into a buffer to be used by MicroEMACS. Then MicroEMACS will appear automatically. In one window you should see the message:

```
3: missing ';'
```

and in the other you should see your source code for **error.c**, with the cursor set on line 3.

If you had more than one error, typing **<ctrl-X>>** would move you to the next line with an error in it; typing **<ctrl-X><** would return you to the previous error. With some errors, such as those for missing braces or semicolons, the compiler cannot always tell exactly which

line the error occurred on, but it will almost always point to a line that is near the source of the error.

Now, correct the error by typing a semicolon at the end of line 2. Close the file by typing **<ctrl-Z>**. **cc** will be invoked again automatically.

cc will continue to compile your program either until the program compiles without error, or until you exit from MicroEMACS by typing **<ctrl-U>** followed by **<ctrl-X><ctrl-C>**.

The MicroEMACS Help Facility

MicroEMACS has a built-in help function. With it, you can ask for information either for a word that you type in, or for a word over which the cursor is positioned. The MicroEMACS help file contains the bindings for all library functions and macros included with COHERENT.

For example, consider that you are preparing a C program and want more information about the function **fopen**. Type **<ctrl-X>?**. At the bottom of the screen will appear the prompt

```
Topic:
```

Type **fopen**. MicroEMACS will search its help file, find its entry for **fopen**, then open a window and print the following:

```
fopen - Open a stream for standard I/O
#include <stdio.h>
FILE *fopen (name, type) char *name, *type;
```

If you wish, you can kill the information in the help window and yank it into your program to ensure that you prepare the function call correctly.

Consider, however, that you are checking a program written earlier, and you wish to check the call to **fopen**. Simply move the cursor until it is positioned over one of the letters in **fopen**, then type **<esc>?**. MicroEMACS will open its help window, and show the same information it did above.

To erase the help window, type **<esc>1**.

Where To Go From Here

For a complete summary of MicroEMACS's commands, see the entry for **me** in the Lexicon. The COHERENT system includes three other editors: the stream editor **sed**, the popular screen editor **vi**, and the interactive line editor **ed**. Each can help you accomplish editing tasks that may not be well suited for MicroEMACS. For more information on these editors, see their tutorials or check their entries in the Lexicon.

Section 11:

nroff, The Text-Formatting Language

nroff is the COHERENT system's text-formatting language. You provide both the text you want formatted and commands to control the formatting; the commands are embedded within the lines of text. **nroff** will then process the text, following the commands that you embedded in the text, and print the formatted text on the standard output.

This tutorial describes how to work with **nroff.** It assumes you are familiar with the basic features of the COHERENT system. In particular, you should know what a *command* is, what a *file* is, and how to create and edit a file. If you are not familiar with these concepts, read *Using the COHERENT System* before you read this tutorial. Relevant Lexicon articles include the one for **nroff**, which summarizes the material in this tutorial, and those for the related program **troff**, **printer** (which summarizes printer-related information), **hpr**, **epson**, and **lpr**.

What is nroff?

nroff is the text processor for the COHERENT system. A *text processor* is a utility that accepts commands and text, and uses the commands to format the text on a page. The commands may call for simple formatting, such as indenting each new paragraph five spaces, to complex formatting of columns and entire pages.

A file that contains text mixed with **nroff** commands is called a *script*. For example, the following **nroff** script

```
.nr Z 0 5
.nf
I tire of love,
.ti \n+Z
I sometimes tire of rhyme;
.ti \n-Z
But money makes me happy
.ti \n+Z
All the time!
.fi
```

produces the following printed text:

```
I tire of love,
      I sometimes tire of rhyme;
But money makes me happy
      All the time!
```

An **nroff** script allows you to change your output very easily. For example, change the minus sign '-' in line 7 of the **nroff** to a plus sign '+', and the formatted text suddenly becomes:

```
I tire of love,
      I sometimes tire of rhyme;
            But money makes me happy
                  All the time!
```

As you can see, **nroff** is a powerful and versatile formatter.

In truth, however, **nroff** is both a text formatter and a text formatting *language*. With **nroff**, you can write your own text-formatting commands to handle automatically the unique requirements of whatever formatting you need.

nroff Input and Output

Input is what you give to **nroff**. *Output* is what **nroff** returns to you. If you simply type

```
nroff
```

then **nroff** accepts input from your keyboard, and prints its output on your screen. For example, if you want **nroff** to process the contents of a file named **script.r**, type the command line

```
nroff script.r
```

nroff then takes the file **script.r**, processes it, and in a few moments it displays the formatted text on your screen. Note that the suffix **.r** is used by convention to indicate that a file contains an unprocessed **nroff** script.

You can save **nroff**'s output by *redirecting* it into another file. For example, you can redirect **nroff**'s processed output of the file **script.r** into the file named **target** by using the following command:

```
nroff script.r > target
```

Printing nroff Output

The COHERENT system's implementation of **nroff** currently can be used with any variety of printer. COHERENT, however, fully supports three varieties of printer: Epson-compatible dot-matrix printers, printers that use the Hewlett-Packard Page Control Language (PCL) (including the Hewlett-Packard LaserJet and DeskJet families of printers), and any printer that has implemented the PostScript page-control language. The following descriptions assume that you have plugged your printer into a parallel port on your computer, and have installed COHERENT correctly so that it can access your printer.

To print **nroff** output on an Epson-compatible printer, use the commands **epson** and **lpr**. For example, to print the **nroff** output that you have directed into file **text.out**, use the following command:

```
epson text.out | lpr
```

Or, you can pipe the output of **nroff** directly into **epson**, as follows:

```
nroff -ms text.r | epson | lpr
```

In the above example, **text.r** is your input, and **-ms** invokes the **ms** package of macros.

To print on a printer that uses PCL, use the commands **hp** and **hpr**. For example, to print the file **text.out** on a PCL printer, use the command:

```
hp text.out | hpr -B
```

The option **-B** to **hpr** suppresses the printing of a banner page. If you wish, you can pipe the output of **nroff** directly into **hp**, as follows:

```
nroff -ms text.r | hp | hpr -B
```

To access a printer that uses PostScript, use the command **hpr**, but do not use the command **hp**. Also, you use must the **-p** switch to **nroff**, which tells it to generate PostScript output. For example, the following command processes file **text.r** into PostScript output, and passes that output to a PostScript printer:

```
nroff -p -ms text.r | hpr -B
```

All of the above commands are described in their respective entries in the Lexicon. The Lexicon article **printer** summarizes information about using printers with the COHERENT system.

nroff Limitations

Because **nroff** is a text-formatting language rather than a text-formatter *per se*, it makes no assumptions about how you want to lay out your page. It does not automatically leave margins at the top and bottom of pages; it does not automatically number pages; it does not automatically format paragraphs. You must use or create a set of formatting commands, called *macros*, to generate these features. This tutorial will teach you how to write macros that can solve nearly every conceivable formatting problem. As you have seen, too, your copy of the COHERENT system comes with a set of predefined macros, the **-ms** macro package.

The ms Macro Package

A macro package called **-ms** is included with your copy of **nroff**. It provides macros to format paragraphs, produce headers and footers (the areas at the top and bottom of pages, respectively), and perform most other page-formatting tasks. **-ms** is easy to use. The command

```
nroff -ms
```

tells **nroff** to accept input from your keyboard, process it using the **-ms** macro package, and print the output on your screen. The command

```
nroff -ms script.r
```

tells **nroff** to process **script.r** with the **-ms** package and print the output on your terminal; while the command

```
nroff -ms script.r >target
```

redirects the output of **nroff** into the file **target**; and

```
nroff -ms script.r | lpr
```

prints the output on the line printer.

Working with the **-ms** macro package is a good way to gain confidence in working with **nroff** commands. Soon you will learn the correct way to encode **nroff** commands in your scripts.

Using this Tutorial

The only way to learn about **nroff** is to use it. You should type all the examples in this tutorial into your computer and observe how they work. You should also alter the example and examine how your changes affect what **nroff** produces. Don't hesitate to experiment! You can learn more from analyzing why something unexpected happens than you can from simply copying an example that works as you were told it would.

The first section describes how to use **nroff** with the **-ms** macro package. The second section describes how to perform sophisticated formatting. For most users, this chapter contains all the information they need to know.

The rest of the tutorial describes how **nroff** actually works with the input text to produce its output. This will teach you how to write your own **nroff** macros for your special word processing needs.

The -ms Macro Package

As explained above, **nroff** is the text formatter for the COHERENT system. You give **nroff** a *script* — that is, text interspersed with commands that control its processing; **nroff**, in turn, formats your text in the manner dictated by your commands.

nroff's most outstanding feature is its flexibility: you can control line length, page offset, page length, paragraph format, beginning- and end-of-page format, and every other aspect of formatting a document.

nroff has built into it a set of basic commands, called *primitives*, that are used to control formatting. A basic formatting function might require several primitives. For example, formatting a new paragraph requires one primitive to force the printing of the fragment of a line left at the end of the previous paragraph; another primitive to skip a blank line; and a third primitive to indent the first line of the new paragraph. If you were to type directly into your script all the primitives required to control every feature of your document, formatting would be a very difficult task, and mistakes would be common.

Fortunately, another feature of **nroff** makes it easier for you to prepare input: **nroff** allows you to bundle together a group of primitives and give the bundle its own name. Such a bundle is called a *macro*. Whenever you want all the commands in that bundle to be executed, you simply insert the name of the macro into the text. For example, you might group the primitives needed to format a paragraph, and call that bundle **PP**. Then, instead of retyping the primitives, all you need to do is insert the command **.PP** before the start of a paragraph.

-ms is a package of macros that are ready for you to use. When you include the option **-ms** on the **nroff** command line, **nroff** automatically uses the the macros that have been defined in the **-ms** package. These macros will take care of setting line length and page length, numbering pages, formatting paragraphs, and all other formatting tasks. You do not need to know how **nroff**'s primitives are used in the macros; you only need to know the names of the macros and what they do, so that you can insert them correctly into your text.

Using the **-ms** package is a good way to become accustomed to preparing input for **nroff**, so that the features of the primitives will not seem so alien when you eventually choose to work with them. When you become familiar with **nroff**, you may wish to your own macro packages, to handle the unique requirements of different types of documents. For now, however, you will find that the **-ms** package will get you up and running with **nroff**.

Text and Commands

nroff input includes both *text* and *commands*. The commands control the processing of the text. **nroff** distinguishes between text and commands by looking at the first character of each input line. If that character is a period or an apostrophe, the line is a *command*; otherwise, it is *text*.

Earlier in this tutorial, you used the **-ms** package to format a text file that had already been prepared for you. To become more accustomed to using **nroff**, try entering the following text into a file that can be formatted later. Use a text editor (either **ed** or MicroEMACS) to create a file named **script2.r** that contains the following text. It is important for this exercise that you break up the lines as they are shown here:

```
London.  Michaelmas Term lately over,
and the Lord Chancellor sitting in
Lincoln's Inn Hall.  Implacable November weather.
As much mud in the streets, as if the waters
had but newly retired from the face of the
earth, and it would not be wonderful to meet
a Megalosaurus, forty feet long or so, waddling
like an elephantine lizard up Holborn Hill.
```

Note that this file contains no commands; every line is a text line. Process the file with the command:

```
nroff script.r | scat
```

The output is piped to **scat** so that it will not all rush past your screen. **nroff** will process the text, and in a moment you will see the following:

```
London.  Michaelmas Term lately over, and the Lord Chancellor
sitting in Lincoln's Inn Hall.  Implacable November weather.  As
much mud in the streets, as if the waters had but newly retired
from the face of the earth, and it would not be wonderful to meet a
Megalosaurus, forty feet long or so, waddling like an elephantine
lizard up Holborn Hill.
```

When you see this example, the spacing will be different; the spacing for the examples in this tutorial is adjusted to conform to the rest of the tutorial text. Notice that **nroff** automatically adjusts the spacing between words to justify the right margin, even though the input text has a ragged right margin. Each output line contains 65 characters, and each output page contains 66 lines.

Now try processing **script.r** again, this time with the **-ms** macro package. Type

```
nroff -ms script.r | more
```

As you can see, **nroff** again adjusted the spacing to keep a strict right margin. Each line was indented with ten leading spaces, followed by 65 characters of text. The pages output by both the **nroff** command and the **nroff -ms** command both contain 66 lines, but the page built with the **-ms** package left blank lines at the top of the page and printed the page number in a blank space at the bottom of the page. When **nroff** constructs its output, it assumes that your printer prints ten characters per inch (Pica, or 10-pitch spacing) and six lines per inch. Given these assumptions, each page of output from **nroff -ms** fits onto an 8.5 by 11 inch page, with an inch of blank space at the top, at the bottom, and on each side.

As this example shows, **nroff** adjusts the spacing between words to keep a strict right margin. When you type in the text, don't worry about the right margin. You must, however, keep a strict left margin, because when **nroff** encounters a line of text that begins with blank spaces, it breaks the line it was working on and begins a new, indented line.

Also, do not hyphenate words; if you do, **nroff** treats each part as a separate "word" (the first ending with the hyphen character), rather than keeping them joined, as you want.

nroff normally interprets as a command every line that begins with a *period* or an *apostrophe*. However, to include an initial apostrophe or period as a literal part of your document, you must place the characters **\&** before the period or apostrophe.

The remainder of this will show you how to use commands in input text to change the appearance of the output. You can control many aspects of the printed document simply by including the appropriate commands within your text.

Command Names

The name of every **nroff** primitive consists of two lower-case letters. Some commands can also include additional information, or *arguments*. For example, **.sp** is the command to leave vertical space between output lines. The command line

```
.sp
```

leaves one space, whereas

```
.sp 2
```

leaves two spaces. The information that follows the command name on the command line is an argument. Each macro defined in the **-ms** macro package is named with one or two upper-case letters. For example, **.PP** is the name of the macro that begins a new paragraph.

Paragraphs

Every time you want to begin a new paragraph, enter the *paragraph* command **.PP**; that is, place the command line **.PP** in the text. To test this macro, enter the following text under the name **script3.r**:

```
.PP
It is a truth universally acknowledged,
that a single man in possession of a good fortune,
must be in want of a wife.
.PP
However little known the feelings or views of such
a man may be on first entering a neighbourhood, the
truth is so well fixed in the minds of the surrounding
families, that he is considered as the rightful
property of some one or the other of their daughters.
```

When you process this text with the command

```
nroff -ms script3.r | more
```

the result resembles the following:

```
    It is a truth universally acknowledged, that a single man in
possession of a good fortune, must be in want of a wife.
```

```
    However little known the feelings or views of such a man may
be on first entering a neighbourhood, the truth is so well fixed
in the minds of the surrounding families, that he is considered as
the rightful property of some one or the other of their daughters.
```

As the output shows, the **.PP** command inserts a blank line before beginning a new paragraph, and indents the first line of the new paragraph by half an inch.

The **-ms** package also provides another paragraph format: the **.IP** command. This macro creates an *indented paragraph*. The **.PP** macro indents only the first line of each paragraph; however, **.IP** indents every line *except* the first. For example,

```
.IP
This is an indented paragraph.
All the lines are indented by
the same amount.
.PP
This is a normal paragraph.
nroff indents the first line
but does not indent the following lines.
```

gives the output

```
        This is an indented paragraph. All the lines are indented
        by the same amount.

        This is a normal paragraph. nroff indents the first line
but does not indent the following lines.
```

Several options are available for the basic **.IP** macro. You can add two *arguments* to it. **nroff** interprets the first argument after the **.IP** as a *tag* to the paragraph, and it interprets the second argument as the amount of indentation you want. For example,

```
.IP A. 8
This is the first line of text.
nroff indents the following lines by the same
amount as the first.
The indent is eight spaces.
The paragraph includes a tag in the indent.
```

produces

```
A.      This is the first line of text. nroff indents the
        following lines by the same amount as the first. The
        indent is eight spaces. The paragraph includes a tag in
        the indent.
```

You must make sure the indent leaves enough spaces for the tag. If the tag contains blank spaces, enclose it within quotation marks. To see how this works, enter the following script under the title **script4.r**:

```
.IP "King Lear:" 16
Is man no more than this?
Consider him well.
Thou owest the worm no silk,
the beast no hide,
the sheep no wool,
the cat no perfume...
Unaccommodated man is no more
but such a poor, bare, forked
animal as thou art.
```

When processed with the command

```
nroff -ms script4.r >script4.p
```

you see:

```
King Lear:    Is man no more than this? Consider him well. Thou
              owest the worm no silk, the beast no hide, the sheep
              no wool, the cat no perfume... Unaccommodated man
              is no more but such a poor, bare, forked animal as
              thou art.
```

As this example shows, this form of the **.IP** macro can be used to format the script for a play.

If you do not want a tag, but merely wish to set the indentation to something other than the default setting of five spaces, then use a pair of quotation marks with nothing between them for the first field:

```
.IP "" 8
```

If you forget the quotation marks, you will not get what you expect: **nroff** will interpret '8' as a tag and use the normal indentation of five spaces.

Once you set the amount of indentation, the new indentation stays in effect until you change it again. For example, if you format a paragraph with

```
.IP "" 8
```

and follow it with another paragraph that begins with **.IP**, **nroff** will also indent the second paragraph by eight spaces. The indentation will remain in effect until you explicitly change it — for example, by beginning a paragraph with

```
.IP "" 6
```

which resets the indent to six spaces.

Normally, **nroff** measures the paragraph indentation from the left margin. Another variation of **IP** allows you to measure the indentation of a new indented paragraph from the left-hand edge of a previous indented paragraph, thus producing *relative indentation*. To do this, enclose the new paragraph between the macros **RS** and **RE** (for **r**elative indent **s**tart and **r**elative indent **e**nd). Copy the following script into the file **script5.r**:

```
.IP
And it came to pass in an eveningtide,
that David arose from off his bed ...
and from the roof he saw a woman washing
herself; and the woman was very beautiful
to look upon.  And David sent and enquired
after the woman.  And one said,
.RS
.IP
Is not this Bathsheba, the daughter of Eliam,
the wife of Uriah the Hittite?
.RE
.IP
And David sent messengers and took her; and
she came in unto him, and he ...
and she returned unto her house.
```

When processed through **nroff** with the command

```
nroff -ms script5.r >script5.p
```

the output resembles the following:

```
And it came to pass in an eveningtide, that David arose from off
his bed ... and from the roof he saw a woman washing herself; and
the woman was very beautiful to look upon.  And David sent and
enquired after the woman.  And one said,

        Is not this Bathsheba, the daughter of Eliam, the wife of
        Uriah the Hittite?

And David sent messengers and took her; and she came in unto him,
and he ... and she returned unto her house.
```

You can include any number of indented paragraphs between **.RS** and **.RE**. Also, you can specify tags and different indents just as for ordinary indented paragraphs. You can even nest **.RS** and **.RE** pairs inside each other to produce multiple relative indents. Just remember that an **.RS** must always be balanced by an **.RE**. Type the following into the file **script6.r** to see how **nroff** handles nested flashbacks:

```
.IP
In England during World War II, a captain tells the
story of his Free French bomber squadron.
.RS
.IP
In the early days of the war, a French ship picks up
five men adrift in a small boat.  One tells of their
life on Devil's Island.
.RS
.IP
A convict tells others of his past.
.RS
.IP
Publication of anti-Nazi material leads to arrest on
false charges.
.RE
.IP
The convicts escape to help France in the war.
.RE
.IP
When France surrenders, the crew overpowers pro-Vichy
officers and heads for England instead of Marseilles.
.RE
.IP
The captain concludes his story as the bombers return
from a mission.
```

When you process this file with the **-ms** package, the output file **script6.p** should resemble the following:

```
In England during World War II, a captain tells the story of his
Free French bomber squadron.

        In the early days of the war, a French ship picks up five
        men adrift in a small boat.  One tells of their life on
        Devil's Island.

                A convict tells others of his past.

                        Publication of anti-Nazi material leads to
                        arrest on false charges.

                The convicts escape to help France in the war.

        When France surrenders, the crew overpowers pro-Vichy
        officers and heads for England instead of Marseilles.

The captain concludes his story as the bombers return from a
mission.
```

As you can see, each **.RE** command peels away the current layer of indentation and moves you into the previous one. To return to an even earlier level, you must input the appropriate number of **.RE** commands before you begin a paragraph.

A third type of paragraph is the *quoted* paragraph. It produces a paragraph that is indented on both on the right side and on the left side, in order to set off a quotation from the surrounding text. To produce such a paragraph, precede it with the **.QS** macro and follow it with the **.QE** macro. To break the quotation into different sections, insert a blank line in the text before each line that you want to begin a new section. For example, type the following example as **script7.r**:

```
Form of Tender of Rescue from Strange Young Gentleman
to Strange Young Lady at a Fire.
.QS
Although through the fiat of a cruel fate, I have been
debarred the gracious privilege of your acquaintance,
permit me, Miss [here insert name, if known], the
inestimable honor of offering you the aid of a true
and loyal arm against the fiery doom which now
o'ershadows you with its crimson wing.  [This form
to be memorized, and practiced in private.]
.QE
Should she accept, the young gentleman should offer
his arm - bowing, and observing "Permit me" -
and so escort her to the fire escape and deposit
her in it.
```

After processing with the **-ms** package, the output file **script7.p** should resemble the following:

```
Form of Tender of Rescue from Strange Young Gentleman to Strange
Young Lady at a Fire.

        Although through the fiat of a cruel fate, I have been
        debarred the gracious privilege of your acquaintance,
        permit me, Miss [here insert name, if known], the
        inestimable honor of offering you the aid of a true and
        loyal arm against the fiery doom which now o'ershadows you
        with its crimson wing.  [This form to be memorized, and
        practiced in private.]

Should she accept, the young gentleman should offer his arm -
bowing, and observing "Permit me" - and so escort her to the fire
escape and deposit her in it.
```

Section Headings

The *section heading* macro **.SH** prints a heading or title. For example:

```
.SH
Section Headings
```

The heading may be more than one line long; consequently, you should follow a section heading with a **.PP** or an **.IP** macro. **nroff** leaves a blank line before the heading and prints the heading flush with the left margin in **boldface** type, as described below in the section on Fonts.

The *numbered heading* macro **.NH** produces consecutively numbered section headings. For example:

```
.NH
Guess What's Coming to Dinner?
.NH
Guess Why I Won't be There?
```

produces

```
1.   Guess What's Coming to Dinner?

2.   Guess Why I Won't Be There?
```

You can number subsection headings by entering a number from two to five to the **.NH** macro. The number indicates the level of section headings; for example, **.NH 2** numbers subsection headings, **.NH 3** numbers sub-subsection headings. For example:

```
.NH
Guess What's Coming to Dinner?
.NH 2
Guess What it Looks Like?
.NH 3
Teeth Like That Might Frighten the Children!
.NH 2
What Does it Eat?
.NH
Guess Why I Won't be There?
```

produces:

```
1.   Guess What's Coming to Dinner?

1.1   Guess What it Looks Like?

1.1.1   Teeth Like That Might Frighten the Children!

1.2   What Does it Eat?

2.   Guess Why I Won't be There?
```

The number on the **.NH** command line is *not* the number that appears before the heading; instead, it controls how many "parts" appear in the number. For example, **.NH 3** produces a three-part number, such as **2.5.3**, whereas **.NH 4** produces a four-part number, such as **7.4.5.2**.

You can reset the entire numbering scheme by using the command **NH 0**; for example,

```
.NH 0
Through The Mandelbrot Set With Rod and Gun
```

produces

```
1.   Through The Mandelbrot Set With Rod and Gun
```

with numbering starting at one.

Title Page

If you want your output to begin with a title page, begin the input with the following.

```
.TL
Title of Document (may be more than one line)
.AU
Name(s) of Author(s) (may be more than one line)
.AI
Institution(s) of Author(s)
.AB
Abstract (line length 5.5 inches)
.AE
```

The **.TL** macro indicates the *title*, the **.AU** macro indicates the *author*, the **.AI** macro indicates the *author's institution*, and the **.AB** macro precedes the *abstract*. The **.AE** macro, for **a**bstract **e**nd, marks the end of the abstract. If you do not want some of these headings to appear, simply omit the relevant macros. Begin the body of the document immediately after the **.AE** macro. The body must begin with a formatting command, such as **.PP** or **.SH.**

Note that the *end abstract* macro **.AE** also prints today's date automatically. To do so, **nroff** reads the date as encoded in the COHERENT system. Before you use these macros, be sure that you have set the correct date in the COHERENT system.

To see how these macros work, type the following script into file **script8.r**:

```
.TL
Tickling in the Therapy of
von Muenchausen's Syndrome
.AU
P. R. Sanserif
.AI
The Department of Parapsychology
The University of Southern North Dakota
at Hoople
.AB
Study of 150 subjects (75 men and 76 women)
indicated that hard tickling may prove beneficial
to patients with von Muenchausen's syndrome.
Applications for a seven-figure grant have been
made to continue research in this area.
.AE
.PP
Due to complications in our experiment, this paper
has now been withdrawn.
```

After processing with the **-ms** macro package, you will see that in the outputfile **script8.p**, **nroff** placed the text on the same page as the title information. You may or may not want this to happen. If you do not, one solution is to insert two additional commands between the **.AE** macro and the body of your text:

```
.PP
.bp
```

begin a new page

Headers and Footers

The *header* macro controls the format of the top of each page. It automatically skips one inch at the top of the page. The *footer* macro controls the format of the bottom of each page. It stops printing text one inch above the bottom of the page, and prints the page number.

It is easy to print either a page header or a page footer. Both the page header and the page footer are three-part titles: **nroff** prints the first part on the left side of the page, the second part in the middle, and the third part on the right side of the page. The parts of the header title are named:

LT: left, top
CT: center, top
RT: right, top

and the parts of the footer title are named:

LF: left, footerf
RF: right, footer

These parts are called *strings*. A later section of this tutorial describes strings in detail. Normally, these strings are undefined, except for **CF**, which prints the current page number; therefore, the header macro normally prints nothing, and the footer macro prints only the page number in the center of the block of space at the bottom of each page. However, you can set any portion of the header or footer to print what you like. To set the left portion of the header, for example, type the following:

```
.ds LT "Walnuts in History"
```

Note that you do *not* type a period before the **LT**. After you define **LT** in this fashion, **nroff** will print

```
Walnuts in History
```

at the top of each page on the left-hand side. If you want the date to appear on the right-hand side of the header, type:

```
.ds RT "\*(Ds"
```

define a string x to have a given value

The string **Ds** is automatically initialized to today's date, as set on your COHERENT system. A later section of this tutorial will present strings in detail. For now, all you need to know is that whenever you want **nroff** to insert today's date into your script automatically, just type the entry ***(Ds**. This entry does *not* have to be at the beginning of a line to work.

Use the same procedure to define the strings in the footer title. If you want something other than the page number to appear in the position allocated to **CF**, use the **.ds** primitive to redefine **CF**. If you want nothing to appear there, type

```
.ds CF ""
```

Wherever you want the current page number to appear in the header or footer, use the symbol '%'. For example, if you want the page number to appear in the upper right-hand corner of each page, type

```
.ds RT "Page %"
```

Be sure to type in all of the macros to define headers and footers *before* you begin to type in your text. Otherwise, your headers and footers will not appear on the first page of the formatted output.

To see how this works, try editing the file **script1.r**. At the top, insert the macro

```
.ds RT "\*(Ds"
```

and reprocess the file using the **-ms** macro package. Each output page should have today's date written in the upper right-hand corner.

Fonts

nroff normally prints ordinary, or "Roman", characters. In addition, **nroff** can print **boldface** and *italic* characters. Each of the three styles of type — Roman, boldface, and italic — is called a *font*, in keeping with typesetting terminology.

nroff prints each boldface and italic character by generating a special three-character output sequence. It prints the boldface character **c**, for example, by printing a 'c', then the backspace character **<ctrl-H>**, and then another 'c'. This sequence emaphasizes 'c' by forcing your printer to print it twice. **nroff** represents an italic character *c* with the underscore character '_', followed by the backspace character **<ctrl-H>**, followed by 'c'.

Because of these special representations, the appearance of **nroff** boldface and italic fonts depends on the device on which you see the output. On your terminal, the **<ctrl-H>** backspaces the cursor, and the third character of each sequence replaces the first; therefore, boldface and italic characters appear the same as Roman characters. On a printer, the appearance depends on the characteristics of the printer. The COHERENT system provides a *filter* or a printer *driver* to print boldface and italic character sequences appropriately on certain devices.

The **-ms** macro package includes three commands for easy printing in specific fonts: the *boldface* command **.B**, the *italic* command **.I**, and the *Roman* command **.R**. To print a single word in boldface, do the following:

```
The last word is printed in
.B boldface.
```

Likewise for italics:

```
The last word is printed in
.I italics.
```

These examples printed a word in a different font. You can print several words in a different font by enclosing the words within quotation marks on the command line:

```
This sentence ends with
.B "three bold words".
```

You can also switch fonts by using one of the font commands with nothing after it on the command line. For example,

```
.B
This entire sentence is printed in boldface.
.R
```

or

```
.I
This entire sentence is printed in italics.
.R
```

In these examples, the Roman font command **.R** is needed to return to the normal font after completing the boldface or italic text.

On rare occasions, you might want different parts of one word to be in different fonts. You cannot use the **-ms** macros to produce mixed-font words directly. A later section of this tutorial gives additional information about **nroff** fonts. As explained there, the input

```
This manual describes \fBnroff\fR's powerful features.
```

produces the output:

```
This manual describes nroff's powerful features.
```

The word **nroff** is boldface but the following apostrophe and 's' are Roman.

Special Characters

A few characters have special meaning to **nroff.** You should be aware of these characters if you want **nroff** to process your text properly.

As mentioned earlier, the period and the apostrophe introduce **nroff** command lines. Each is a special character if it is the *first non-space character* on an input line. If you wish to use a period or an apostrophe at the start of an input line simply as part of your text, you must precede it with a backslash and ampersand "**\&**". For example, the input

```
The footnote command
.DS
\&.FT
.DE
generates footnotes for you automatically.
```

produces the output

```
The footnote command

.FT

generates footnotes for you automatically.
```

Neither the period nor the apostrophe is a special character unless it is the first non-space character on a line.

The most important special character for **nroff** is the backslash '\'. It changes the meaning of the following character or characters. If you simply want a backslash to appear as part of your text, you must follow it with the letter 'e'; that is, use "\e" in your input to have '\' appear in your output. Later sections of this tutorial describe other special uses for backslash.

Footnotes

You can place footnotes between the *footnote start* command **.FS** and the *footnote end* command **.FE**, as in the following example:

```
.FS
*MicroKVETCH Electronic Nag is a
copyrighted trademark of Caveat Emptor
Software, Inc.
.FE
```

You should insert each footnote into your text where the reference to it occurs; **nroff** will see to it that the footnote appears at the bottom of the correct page. Footnotes should be inserted as follows:

```
The notion that we have been visited
by visitors from outer space may seem
outlandish(1)
.FS
1. Raucus J, O'Hooligan R: "Viruses
from Venus?" \fIJ Earth Med Assoc\fR,
1985;36:412-414.
.FE
but reason compels us to exclude no ...
```

The journal article cited in the footnote will appear at the bottom of the page, with the journal name in italics.

Displays and Keeps

A *display* is a portion of text, such as a graph or a table, that should appear in the output exactly as it is typed in the input. **nroff** normally alters the spacings between elements in your text, which, of course, would destroy the appearance of a display. Therefore, **nroff** has macros to tell it that a portion of text is a display, and so not to alter spacings between elements or split it between two pages. These macros are the *display start* macro **.DS** and the *display end* macro **.DE**. You should your display between these macros, as follows:

```
.DS
The text of the display goes here,
exactly
as
you
want
it
to appear in the output.
.DE
```

The **.DS** macro comes in three varieties. The *display start centered* macro **.DS C** centers every line of your display. Because **nroff** centers each line individually, both right and left margins are ragged. The *display start block-centered* macro **.DS B** takes the entire display at once and centers it. You can think of this as simply shifting the display to the right by an appropriate amount. The *display start indented* macro **.DS I** indents the entire display by half an inch.

If your display is longer than one page, do not use **.DS** or any of its variants. Instead, begin the display with one of the following.

The *centered display* macro **.CD** centers each line of the display. The *block-centered display* macro **.BD** considers the entire display as a block and centers it. The *left display* macro **.LD** performs no indenting or centering, but simply begins each line at the left margin. Finally, the *indented display* macro **.ID** indents each line by half an inch. If you begin the display with one of these macros, do *not* end it with **.DE**; rather, just type **.PP** or **.SH** or whatever other macro is needed at that point.

To see how displays work, type the following into the file **script9.r** and process it with the **-ms** macro package:

```
.PP
.DS C
Tyger! Tyger! burning bright
In the forests of the night,
What immortal hand or eye
Could frame thy fearful symmetry?
Burma Shave
.DE
```

When the output file **script9.p** is read, the results will appear as follows:

```
Tyger! Tyger! burning bright
In the forests of the night,
What immortal hand or eye
Could frame thy fearful symmetry?
Burma Shave
```

You must remember one important fact when you use display macros: the normal length of output lines is 6.5 inches, but if the display contains lines longer than this **nroff** simply prints them as they are. If a line is too long to fit onto the page, what occurs afterwards depends upon the output device. If you are displaying the output on the screen, the text will be displayed as far as possible to the right, then the remainder will be wrapped around onto the next line, without indentation. On most printers, however, the chunk of text that extends past the right margin will simply be lopped off and thrown away. In any event, the effect is usually quite unsightly. The only restriction on what you can safely put in a display, then, is that lines should be no longer than 6.5 inches. If you are using an indented display, lines should be no longer than six inches.

A *keep* is a display macro: you put text between the *keep start* macro **.KS** and the *keep end* macro **.KE** when you want it all kept on the same page. If you put a block of text between these macros that proves to be longer than one page, **nroff** moves the excess text onto a new page.

The major difference between the keep and the display is that normal processing occurs in the keep: **nroff** adjusts spacings between words, hyphenates words, justifies lines, and performs all other formatting tasks, just as it normally does.

Other Commands

Several of **nroff**'s primitives can be used with the **-ms** macro package. The primitive

```
.sp N
```

skips *N* lines on the output page; for example, **.sp 4** skips four lines.

The *begin page* primitive **.bp** tells **nroff** to begin a new page, no matter where it is on the current page.

The remaining sections of this tutorial provide more information about these other **nroff** primitives.

Introducing nroff's Primitives

The rest of this tutorial describe **nroff**'s *basic commands* — the commands that are "built into" **nroff**, and from which macros are assembled. These basic commands, or *primitives*, form **nroff**'s text formatting language. Once you have mastered the primitives, you will be able to write macros to control automatically even the most difficult text formatting tasks.

The rest of this tutorial includes a number of exercises. You should type them into your system and execute them as described in the tutorial; this will greatly increase the rate at which you master **nroff**. None of the following examples should be processed with the **-ms** macro package; the purpose of this portion of the tutorial is to teach you how to create your own text processing routines, rather than how to use ones that have already been written.

Page Format

When deciding how to process text, you must first decide how to position the text on the printed page. You must control line length, left and right margins, page offset (i.e., how far from the left edge of the page each line begins), and page length. Controlling these functions is quite easy with the appropriate **nroff** commands.

The *line length* primitive **.ll** controls the line length; and the *page offset* command **.po** controls the page offset. If you are writing an **nroff** script, you should include these commands before the beginning of your text, so that **nroff** can put them into effect immediately. The following example uses a line length of three inches and a page offset of two inches. Type this into your system under the name **ex1.r**. Note, by the way, that the text to the right of the characters '\"' is a *comment*, and there is no need for you to type it into your system:

```
.ll 3i        \" set line length
.po 2i        \" set page offset
Along outside of the front fence ran the country
road, dusty in the summertime, and a good place for
snakes -- they liked to lie in it and sun themselves;
when they were rattlesnakes or puff adders, we killed
them; when they were black snakes, or racers, or belonged
to the fabled "hoop" breed, we fled, without shame; when
they were "house snakes", or "garters", we carried them
home and put them in Aunt Patsy's work basket for a
surprise; for she was prejudiced against snakes, and
always when she took the basket in her lap and they
began to climb out of it it disordered her mind.
```

Process this script by typing the command

```
nroff ex1.r >ex1.p
```

From this point on, you should *not* use the **-ms** macro package with your **nroff** examples. When you display the output stored in the file **ex1.p**, you will see that the length of each line is three inches, and each line begins two inches from the left-hand margin.

As you noticed, line length and page offset were set in *inches*. **nroff** output can be controlled using a number of different units of measurements, including inches, number of characters, or lines, or *machine units*. A following section discusses **nroff** units of measurement in detail.

As noted above, this example contains two *comments*. **nroff** ignores any text that appears on a line after "\"". You should use comments, for the benefit of anyone who must read your **nroff** script (including yourself). The above example used the comments

```
\" set line length
\" set page offset
```

to help you understand the **.ll** and **.po** commands. Judicious comments can make a complex script much easier to understand.

Breaks

Before you look at the *break* primitive **.br**, it is helpful to examine how **nroff** constructs a finished line of output. Suppose, for example, that you tell **nroff** that you want each output line to be five inches long. **nroff** takes your input one word at a time, and attempts to squeeze that word into the space that has not yet been taken up in the line. When **nroff** finally picks up a word that is to large to fit into the amount of space left in the line, it either puts the word aside entirely, or hyphenates the word and places the hyphenated portion into the line. **nroff** then inserts extra blank spaces between the words to justify the line. The *break* primitive **.br**, however, tells **nroff** to print whatever words have already been put into the line, even if they do not form a complete line, and without performing right justification.

The idea of a break might seem strange at first, but you are familiar with a simple example: the end of a paragraph. You do not want the start of a new paragraph to be on the same line as the end of the previous paragraph: you want to print the end of the previous paragraph whether or not it fills a complete line; and you want to begin the new paragraph on a new line. As you will learn later, some **nroff** commands cause breaks automatically; you should be aware of this when you use them.

Fill and Adjust Modes

Two terms describe how **nroff** processes your input to create its output: *filling*, and *adjusting* or *justifying*. Unless you order it not to, **nroff** operates in the *fill* and *adjust* modes. The *no-fill* primitive **.nf** tells **nroff** to stop using fill mode. The *fill* primitive **.fi** tells it to resume using the fill mode. In a similar way, the *adjust* primitive **.ad** tells **nroff** to use adjust mode, whereas the *no adjust* primitive **.na** tells it to use no-adjust mode.

As mentioned above, **nroff** by default is in both fill mode and adjust mode, so you do not need to begin your script with **.fi** and **.ad** if you want **nroff** to fill and adjust your text. However, if you turn off filling and adjusting by using the **.nf** and **.na** commands, you must use the **.fi** and **.ad** commands to turn filling and adjusting back on.

When you use **.nf** to turn off fill mode, **nroff** no longer tries to fill lines to a fixed line length. It prints each line of input text exactly as received. However, a sufficiently long line of text run off the right-hand edge of the page if **nroff** were to print it as entered. If the input line cannot fit on one line, **nroff** prints as much as it can fit on one line, then breaks the line and prints the rest on the next line with no page offset.

In adjust mode, **nroff** inserts extra spaces between words to justify lines of text, as described above. When **nroff** is in no-fill mode, it is automatically in no-adjust mode: with no fixed line length, there is no need to insert extra spaces. *Moral:* you can fill without

adjusting, but you cannot adjust without filling.

If you request filling but not adjusting, **nroff** fills the output line as described earlier, but does not insert extra spaces between words; that is, it does not try to keep an even right margin. Every output line either is shorter than the line length you specified, or exactly as long.

The **.ad** primitive includes several options. If you use the command **.ad** without an argument, **nroff** keeps strict left and right margins. The primitive **.ad l** justifies the left margin only; **.ad r** justifies the right margin only; and **.ad b** justifies both margins (this, of course, is the default). Finally, **.ad c** centers output lines while keeping their lengths less than or equal to the line length, as set with the **.ll** command.

Remember that **nroff** ignores adjustment requests if you are in no-fill mode. If **nroff** is in fill mode and you request any variety of adjustment, it adjusts accordingly until you issue either a no-fill or a no-adjust command. If you give a no-fill command, only a fill command restores adjustment; any plea for a different kind of adjustment is ignored while **nroff** is in no-fill mode.

To see how this works, type the following script under the name **ex2.r**, and process it as above:

```
.ll 3.75i
.sp          \" space
When we were alone, I introduced the subject
of death, and endeavored to maintain that the fear
of it might be got over.  I told [Johnson] that
David Hume said to me, he was no more uneasy to
think that he should not be after this life, than
that he had not been before he began to exist.
.sp
.na          \"no adjust
JOHNSON:  "Sir, if he really thinks so,
his perceptions are disturbed;
he is mad: if he does not think so, he
lies .... When he dies, he at
least gives up all he has."
.sp
.ad r        \"right-adjust
BOSWELL:  "Foote, sir, told me that
he was not afraid to die."
.sp
.nf          \"no-fill
JOHNSON:  "It is not true, sir.
Hold a pistol to Foote's
breast or to Hume's breast,
and threaten to kill them,
and you'll see how they behave."
.sp
.fi          \"fill
BOSWELL:  "But may we not fortify our minds for
the approach of death?"
.sp
JOHNSON:  "No, sir, let it alone.  It matters not
how a man
dies, but how he lives.  The act of dying is not of
```

```
importance, it lasts so short a time .... A man
knows it must be so, and submits.
It will do him no good to whine."
```

When you process this input with **nroff**, your output should look like this:

```
When we were alone, I introduced the subject of death, and
endeavored to maintain that the fear of it might be got over. I
told [Johnson] that David Hume said to me, he was no more uneasy to
think that he should not be after this life, than that he had not
been before he began to exist.

JOHNSON: "Sir, if he really thinks so, his perceptions are
disturbed; he is mad: if he does not think so, he lies .... When he
dies, he at least gives up all he has."

BOSWELL: "Foote, sir, told me that he was not afraid to die."

JOHNSON:  "It is not true, sir.
Hold a pistol to Foote's
breast or to Hume's breast,
and threaten to kill them,
and you'll see how they behave."

    BOSWELL: "But may we not fortify our minds for the approach of
death?"

   JOHNSON: "No, sir, let it alone. It matters not how a man dies,
  but how he lives. The act of dying is not of importance, it lasts
     so short a time .... A man knows it must be so, and submits. It
will do him no good to whine."
```

After the **.na** primitive, **nroff** fills but does not adjust the second paragraph. After **.ad r**, it fills and right adjusts the third paragraph. After **.nf**, it neither fills nor adjusts the fourth paragraphs. Finally, after **.fi**, it fills the fifth and sixth paragraphs and uses the **.ad r** adjust option that was in effect previously.

Under certain extreme conditions, **nroff** cannot adjust a line even though it is in adjust mode. If, for example, you specified a line length of one inch, a seven-letter or eight-letter word would then take up most of a line. When such a word was then followed by a word that could not fit into the line after it, **nroff** would begin a new line with the second word rather than violate the right margin by inserting the into the line. When a line has only one word in it, **nroff** obviously cannot adjust the line by inserting extra spaces between words; therefore, the right margin is left uneven, as though **nroff** were in no-adjust mode.

Defining Paragraphs

What happens if you copy text from several pages of a book into a file without adding any formatting commands, and then process the file with **nroff**? There is no page offset, because **nroff**'s default page-offset setting is zero; and the processed lines are set to the default length of 6.5 inches (65 Pica characters).

More interesting things happen with paragraphs. Suppose you skip a line between paragraphs and begin each paragraph by indenting five spaces. The blank line in the input text causes a break, and forces **nroff** to print a blank line. The last line of each paragraph is unadjusted, and a blank line appears before the next paragraph. Initial blank spaces in a line of input also cause a break. In this example, the breaks caused by initial blank spaces at the beginning of each paragraph do nothing, because the preceding blank line forces out the last line of the preceding paragraph. **nroff** always considers initial blank spaces in a

line to be significant, and preserves them in the output.

To see how blank lines and initial spaces affect **nroff**'s output, copy the following example and run it through **nroff**:

```
     Here is a little text so you can see
whether nroff will ignore the initial
indentation
            in this very very long sentence.
Here is a little bit more text.

     And here is something to mimic
the beginning of a new paragraph.
```

The output should look like this:

```
   Here is a little text so you can see whether nroff will ignore
the initial indentation
       in this very very long sentence. Here is a little bit more
text.

   And here is something to mimic the beginning of a new
paragraph.
```

Instead of leaving a blank line in the text, you could use the *space* primitive **.sp 1**, which causes a break and inserts one blank line into the output. In a similar way, **.sp 5** causes a break and inserts five blank lines in the output. Edit the example and replace the blank line with the command line:

```
.sp 1
```

You will see that it has the same effect. You can also use the form **.sp**; **nroff** assumes you want one space if you omit the argument.

Most **nroff** input consists of many paragraphs that contain text, and you probably want each paragraph to have the same format in the output. Rather than formatting each paragraph explicitly, as in this example, you can use the *macro* facility of **nroff** to define a sequence of commands to format a paragraph. Macros are described in detail later in this tutorial.

Centering

The *center* primitive **.ce** centers one or more lines of text. For example, you can center a two-line heading as follows:

```
.ce 2
Heading Printed
In Center of Page
```

If you use the **.ce** command with no argument, **nroff** assumes a default argument of one, and centers only the next line of input. The command **ce 0** cancels any earlier centering command that is in operation.

Tabs

If your **nroff** input includes tables, you may find it convenient to use tabs to separate items in a line of the table. **nroff** recognizes the **<tab>** character and expands it into spaces. If you use tabs to format a table, remember to use no-fill mode; otherwise, **nroff** tries to fill and adjust your output lines.

By default, **nroff** sets a tab stops after every eight characters. You can use the *tab* primitive **.ta** to change the positions of the tab stops. For example,

```
.ta 10 20 30 40 50 60
```

sets tab stops ten characters apart rather than eight. **.ta** can also be used to fix tab stops in inches rather than after a number of characters; for example,

```
.ta 0.8i 2.0i
```

sets tab stops after 0.8 inches and 2.0 inches on the output line. This is quite helpful when you are designing a table.

You can use the *tab character* command **.tc** to change the character **nroff** prints between its current position and the next tab stop. Enter the following text to see how this primitive works:

```
.ta 9 19 29 39
.tc *
.nf
<tab>1<tab>2<tab>3<tab>4
```

The output file, **ex3.p**, should appear as follows:

```
*********1*********2*********3*********4
```

Page Breaks

The *begin page* primitive **.bp** causes a break and forces **nroff** to the next output page. By default, **nroff** assumes a page length of 11 inches (66 lines). You can change the page length with the *page length* command **.pl.** For example,

```
.pl 2i
```

specifies a two-inch page length.

handles

At this point, the question arises about how **nroff** top and bottom page margins, number pages, and other and other aspects of page layout. The answer is that **nroff** merely keeps track of the current output page number and the current line number on the current output page; designing top and bottom margins, page headers and footers, and other aspects of page layout is up to you.

Can **nroff** execute a set of commands whenever it reaches a certain position on the page? This would solve the problem of producing top and bottom margins, and you would not have to guess where to insert the commands in your script. In fact, you can tell **nroff** to do this, by using *traps*. The next section of this tutorial describes macros and traps and how to use them to format a page.

Macros and Traps

This section presents **nroff** macros: how to write them, how to tell **nroff** to execute them at a give point on every output page, and how to install a macro file under the COHERENT system

As with previous sections, this one uses a number of exercises. Working the exercises will help you master **nroff** quickly. When you format the exercise scripts, do not use the **-ms** option. Also, it is not necessary for you to copy the comments into your system; they are here to help you understand what each **nroff** command does, but they have no effect on how the script executes.

What Is a Macro?

To become familiar with the idea of a *macro*, consider the problem of formatting a paragraph. Whenever you come to a new paragraph, you want **nroff** to skip a line and indent the first line five spaces. Because **nroff** preserves blank lines and initial indentations, you could force **nroff** to break your text into paragraphs simply by inserting a blank line and spaces directly into your text. The same effect, however, can be achieved by inserting the following set of **nroff** commands

```
.br             \" break
.sp             \" skip a line
.ti 5           \" indent next line 5 spaces
```

between the end of each paragraph and the start of the next paragraph. You should recognize the first two commands: **.br** causes a break, so that **nroff** prints the last line of the previous paragraph even though it might not be a complete line; **.sp** skips a line before the next paragraph begins. The third command is the *temporary indent* command **.ti**, which tells **nroff** to indent the next line; the number indicates how many spaces to indent. The following exercise, **ex4.r**, demonstrates how this works:

```
.ll 3i          \" line length
.po 3i          \" page offset
.ti 5           \" indent next line
Adam was human--this explains it all.  He did
not want the apple for the apple's sake, he
wanted it because it was forbidden.  The mistake
was in not forbidding the serpent; then he would
have eaten the serpent.
.br             \" break
.sp             \" skip a line
.ti 5           \" indent next line
Training is everything.  The peach was once a bitter
almond; cauliflower is nothing but cabbage with a
college education.
.br
.sp
.ti 5
Habit is habit, and not to be flung out of the window
by any man, but coaxed downstairs a step at a time.
```

After you have processed this file, the output file **ex4.p** should resemble the following:

```
    Adam was human--this explains it all. He did not want the
apple for the apple's sake, he wanted it because it was forbidden.
The mistake was in not forbidding the serpent; then he would have
eaten the serpent.

    Training is everything. The peach was once a bitter almond;
cauliflower is nothing but cabbage with a college education.

    Habit is habit, and not to be flung out of the window by any
man, but coaxed downstairs a step at a time.
```

Now, in a small file it would be easy to type all of the **nroff** primitives directly into your input text; however, what if your file is very long, with hundreds of paragraphs? Every time you wanted to begin a paragraph, you would have to include that set of commands within the text. You would save considerable agony if you could bundle these commands together under a common *name*; then you could simply put that *name* into your text whenever you wanted **nroff** to perform these commands, rather than typing the commands themselves over and over again.

As you probably have guessed by now, you can do just that; the set of commands is called a *macro*. The following shows the selections from Pudd'nhead Wilson's calendar set with a macro called **.PP** that takes care of formatting each paragraph. The following exercise, **ex5.r**, shows how to bundle together the **nroff** primitives for formatting paragraphs into the **.PP** macro:

```
.de PP          \" define the PP macro
.br             \" break the line
.sp             \" insert a blank line
.ti 5           \" indent next line 5 spaces
..              \" two periods ends the macro definition
.PP             \" execute PP macro
Adam was human--this explains it all.  He did
not want the apple for the apple's sake, he
wanted it because it was forbidden.  The mistake
was in not forbidding the serpent; then he would
have eaten the serpent.
.PP
Training is everything.  The peach was once a bitter
almond; cauliflower is nothing but cabbage with a
college education.
.PP
Habit is habit, and not to be flung out of the window
by any man, but coaxed downstairs a step at a time.
```

As you can see, using a macro can save you a considerable amount of work when you prepare your script.

Introducing Traps

Now, consider the problem of formatting the beginning and ending of each page of output. You could define what are traditionally called *header* and *footer* macros, which contain the commands you want performed at the top and bottom of each page. However, how can you tell **nroff** when to execute these macros? You cannot possibly know where to call these macros in the input text, because you cannot know where any given text line will appear in the output until you have processed it through **nroff**. This problem is solved by using

traps.

nroff keeps track of its vertical position on each output page. You can set a *trap* that tells **nroff** to execute a macro at a particular vertical position on every page. When a line of output reaches or extends past the position that is specified in your *trap*, **nroff** then executes the commands named in the trap command before processing any more input text.

You can set a trap by using the *when* command **.wh**. For example, if you want **nroff** to call your header macro **.HD** at the very top of each page, the command

```
.wh 0 HD          \" set header trap
```

sets a trap for the macro **.HD** at vertical position 0 (the very top of the page) of every output page. The macro **.HD** will then be executed every time **nroff** begins a new page. To have your footer macro **.FO** execute one inch from the bottom of each page, use the command

```
.wh -1i FO        \" set footer trap
```

The negative number tells **nroff** to measure distance from the *bottom* of the page rather than from the top; the **i** is an abbreviation for inches. (**nroff** recognizes various units of measurement; this will be described in more detail later.)

Headers and Footers

Suppose you want to design the output page by defining the header and footer macros. A simple header macro merely skips an inch of space at the top of each page; a simple footer macro forces printing to stop an inch from the bottom of each page and prints the page number. **nroff** does not print page numbers automatically, but it does automatically keep track of which output page it is on. It stores the page number internally in a *number register* that you can access with the symbol '%'. (A later section gives more information about number registers and how to use them.)

The following gives a simple footer macro that prints the page number:

```
.de FO            \" define footer macro FO
'sp 4v            \" skip four vertical lines (no break)
.tl ''- % -''     \" print hyphen, page number, hyphen
'bp               \" jump to new page
..                \" end macro definition
```

There are several points of interest raised by this macro.

First, notice that some commands are preceded with an apostrophe rather than with a period. The use of the apostrophe instead of the period tells **nroff** to suppress the break these commands normally cause. You might run into problems if you define your header macro as follows:

```
.de HD            \" header macro
.sp 1i            \" skip an inch (break)
..
```

You want this to leave a blank space of one inch at the top of each page; however, the **.sp** command causes a break, so that if a word were left over from the last line on the preceding page, **nroff** would print it at the very top of the next page. The effect would be quite unsightly. However, if you use **'sp** instead of **.sp** in the macro, **nroff** suppresses the break and does not print the partial word until *after* it performs the macro commands. The same is true for the footer macro: you do not want anything unplanned to be printed in the blank space at the bottom of the page. You should always be conscious of these considerations

when you use commands that cause breaks.

Another new item in the above example is the *title* command **.tl**, which prints a three-part title. A three-part title contains a left part (aligned to the left margin of the page), a center part (centered), and a right part (aligned to the right margin). The command name **.tl** is followed by four apostrophes: **nroff** prints the characters between the first two apostrophes as the left part of the title line, those between the second and third apostrophes as the center part, and those between the third and fourth apostrophes as the right part of the three-part title. If you do not want **nroff** to print anything in one of these positions, simply put nothing between the appropriate pair of quotes. In the above example, the **.tl** primitive tells **nroff** to print nothing in the left and right portions of the footer title line, but to print the page number in the center. If you want an apostrophe to appear as a part of the title, precede it with the backslash character '\'.

nroff considers the length of the title line to be independent of the length of normal output lines; therefore, you must set it with the *length of title* primitive **.lt** unless you want **nroff** to use the default title length of 6.5 inches. For example, to set the length of the title to five inches, use the command

```
.lt 5i
```

In light of all you now know, you should give Pudd'nhead Wilson's calendar the treatment it deserves:

```
.ll 3i              \" set line length to 3 inches
.po 2i              \" set page offset to 3 inches
.pl 9i              \" set page length to 9 inches
.wh 0 HD            \" set the header trap
.wh -1i FO          \" set the footer trap
.de HD              \" define header macro HD
'sp 1i              \" skip 1 inches of space
..                  \" end macro definition
.de FO              \" define footer macro
'sp 2               \" skip 2 lines
.tl ''- % -''       \" define footer title
'bp                 \" begin new page
..                  \" end macro definition
.de PP              \" define paragraph macro
.sp 1               \" skip 1 line of space
.ti 5               \" indent the first line 5 characters
..                  \" end macro definition
.PP
Adam was human--this explains it all.  He did
not want the apple for the apple's sake, he
wanted it because it was forbidden.  The mistake
was in not forbidding the serpent; then he would
have eaten the serpent.
.PP
Training is everything.  The peach was once a bitter
almond; cauliflower is nothing but cabbage with a
college education.
.PP
Habit is habit, and not to be flung out of the window
by any man, but coaxed downstairs a step at a time.
```

As a point of technique, always set header and footer traps early in your input script; otherwise, **nroff** may not print the header on the first page.

Macro Arguments

You can affect how macros function by passing them modifiers, called *arguments*. An argument may be a bit of text that is arranged in a special way by the macro, or it may be a number or other parameter that dictates exactly what the macro does.

As an example of how a macro can handle arguments, consider a macro to format the list of ingredients for a recipe. You want the ingredients to be printed as follows:

```
3 cups of pumpkin
1 cup of milk
1 cup of sugar
1 tsp of ground ginger
1 tbl of cinnamon
```

Each of these lines has the same format: the amount of ingredient, the unit of measurement, the word "of", and the name of the ingredient. You can create a macro (call it **.RE**, for **re**cipe) that encodes the format of these lines and contains three "slots": one slot for the amount, one for the unit of measurement, and one for the name of the ingredient. Each time you use the macro, you indicate what you want to go into each slot, and **nroff** substitutes it for you. The macro **.RE** can be constructed as follows:

```
.de RE              \" define macro RE
\\$1 \\$2 of \\$3 \" set RE's arguments
..                  \" end definition
Here is some text.
.nf                 \" don't fill the recipe
.RE 3 cups pumpkin
.RE 1 cup milk
.RE 1 cup sugar
.RE 1 tsp "ground ginger"
.RE 1 tbl cinnamon
.fi                 \" resume filling
Here is some more text.
.bp                 \" begin a new page, to force printing
```

When you call a macro that takes arguments, the arguments must appear on the same line as the macro command itself. A macro may have up to nine arguments; they are denoted by **\$1**, through **\$9**, respectively: the first field after the macro name is called **\$1**, the second **\$2**, and so on.

If you want to use as an argument a string of characters that includes blank spaces, you must enclose the string within quotation marks, as with the words "ground ginger", in the example above. If you forget to include the quotation marks, **nroff** regards each word in the string as a separate argument, and treats them accordingly.

Note that macros that are called by traps cannot accept arguments.

Double vs. Single Backslashes

If you carefully examine the definition of **RE**, you will see that it identifies each argument with two backslashes:

```
\\$1 \\$2 of \\$3
```

Whenever you identify an argument within a macro, always preface it with two backslashes, rather than one. The reason is that **nroff** in effect processes a macro *twice*: when it first reads it, and later when you call it within your text. Prefacing an argument with *one* backslash tells **nroff** that you want to expand that argument when the macro is first read; prefacing it with two backslashes tells **nroff** that you want to expand it when the macro is called in your text. In nearly every circumstance, you want to expand the arguments in your text, so you should use two backslashes. As you will see, this rule also applies to the use of *strings* and *number registers* within macros.

To see how this works, consider again the **.RE** macro:

```
.de RE
\\$1 \\$2 of \\$3
..
Here is some text.
.nf
.RE 3 cups pumpkin
.RE 1 cup milk
.RE 1 cup sugar
.RE 1 tsp "ground ginger"
.RE 1 tbl cinnamon
.fi
Here is some more text.
.bp
```

Using two backslashes, as above, allows you to redefine what **$1**, **$2**, and $3 mean many times throughout your text, to generate the following output:

```
Here is some text.
3 cups of pumpkin
1 cup of milk
1 cup of sugar
1 tsp of ground ginger
1 tbl of cinnamon
Here is some more text.
```

If you used only one backslash, however, your output would appear as follows:

```
Here is some text.
  of
  of
  of
  of
  of
Here is some more text.
```

nroff could not expand the argument calls (**\$1** etc.), because you had not yet defined them; therefore, it threw them away; and because all of the argument calls had been thrown away, **nroff** then threw all the arguments away. All that was left was word **of**.

Designing and Installing Macros

Now that you have been shown how to write a macro, the next step is to design some macros and install them, so you can call them over and over again.

The first step in designing a macro is to analyze the problem that you want to solve. Suppose that in this instance you want to print a list of names. Each name will consist of a first name, a last name, and the department with which he is associated, and the list will be printed in columns; for example:

```
Firstname     Lastname      Department
```

Moreover, you want to be able to switch the order in which the columns appear without having to retype your list; for example:

```
Lastname      Firstname     Department
```

or

```
Department    Lastname      Firstname
```

In effect, then, you want three macros: one for each of the three orders of columns shown above.

When you have finished designing your macros, they should look something the following. Type the following into the file **tmac.lst**; note that the symbol **<tab>** represents a *tab* character, and should not be entered literally:

```
.\" List macros. $1 represents first name,
.\" $2 last name, $3 department
.de LA
.nf
.ta 1.5i 2.75i
\\$1<tab>\\$2:<tab>\\$3
.Rt
..
.de LB
.nf
.ta 1.5i 2.75i
\\$2,<tab>\\$1:<tab>\\$3
.Rt
..
.de LC
.nf
.ta 1.5i 2.75i
\\$3:<tab>\\$2,<tab>\\$1
.Rt
..
```

The first lines are *comments*, so that anyone who looks at these macros will know what they do. The first command line, introduced with the **.de** command, names each macro. These names were selected after checking the file **tmac.s**, which is where the **-ms** macro package is kept, to confirm that they are not used elsewhere. Naturally, using the same macro name in two different places can lead to a great deal of trouble.

The next command, **.nf**, turns off the **nroff**'s normal right justification, which otherwise would smear a table. The **.ta** command sets the tab characters at certain points on the page, measured from the left margin.

The next line gives the order in which the arguments appear. The arguments are separated by tab characters, and punctuation is inserted. The last command, **.Rt**, calls a macro in the file **tmac.s**; this macro resets **nroff** to its normal fill mode and returns the tab settings to normal. Note that these macros can be used only when you also use the **-ms** macro package.

After you have typed the macros into **tmac.lst**, carefully read over what you type to ensure that no there are no errors; if you find any, be sure to correct them. The final step is to move **tmac.lst** into the directory **/usr/lib**, which is where **tmac.s** is also kept.

To test your new macros, type the following text into the file **ex6.r**:

```
The following lists give the personnel who are involved in
this project:
.sp
.LA Ivan Sanderson Engineering
.LA Marian Maddux Design
.LA George Sutcliffe Electrical
.LA Catherine Williams "Metal Shop"
.LA Fred Wilson Carpentry
.LA Anne Bilecki "Machine Shop"
.sp
.LB Ivan Sanderson Engineering
.LB Marian Maddux Design
.LB George Sutcliffe Electrical
.LB Catherine Williams "Metal Shop"
.LB Fred Wilson Carpentry
.LB Anne Bilecki "Machine Shop"
.sp
.LC Ivan Sanderson Engineering
.LC Marian Maddux Design
.LC George Sutcliffe Electrical
.LC Catherine Williams "Metal Shop"
.LC Fred Wilson Carpentry
.LC Anne Bilecki "Machine Shop"
.sp
We expect that they will receive your full cooperation.
```

The same set of names is used three times; the only difference is the macro call employed.

Now, process this file with the following command:

```
nroff -ms -mlst ex6.r >ex6.p
```

As you can see, when you installed **tmac.list** into **/usr/lib**, you could invoke it in the same way that you invoke **tmac.s** with **-ms**.

When you look at the output file **ex6.p**, you should see something that resembles the following:

```
The following lists give the personnel who are involved in this
project:

Ivan            Sanderson:      Engineering
Marian          Maddux:         Design
George          Sutcliffe:      Electrical
Catherine       Williams:       Metal Shop
Fred            Wilson:         Carpentry
Anne            Bilecki:        Machine Shop

Sanderson,      Ivan:           Engineering
Maddux,         Marian:         Design
Sutcliffe,      George:         Electrical
Williams,       Catherine:      Metal Shop
Wilson,         Fred:           Carpentry
Bilecki,        Anne:           Machine Shop
```

```
Engineering:    Sanderson,      Ivan
Design:         Maddux,         Marian
Electrical:     Sutcliffe,      George
Metal Shop:     Williams,       Catherine
Carpentry:      Wilson,         Fred
Machine Shop:   Bilecki,        Anne

We expect that they will receive your full cooperation.
```

As you grow proficient in writing **nroff** macros, you will probably find it most efficient to keep special macros in their own files; this will save time by ensuring that **nroff** does not have to process macros that are never called.

Strings

Suppose you are writing a script for **nroff** and, to relieve the tedium, decide to punctuate the text occasionally with a rousing cry of "FOOD FIGHT!!". If you plan to interject this phrase more than a few times in your script, you can take advantage of another labor-saving device, called a *string*. You can use a string name as an abbreviation for a long string of characters you use frequently. Like a macro, a string is a *name* that **nroff** associates with a *definition* that you supply. Wherever you put the name in your text, **nroff** prints the definition. Although macros refer to sets of *commands* that you define, strings refer to strings of *characters* that you define.

You define a string with the *define string* primitive **.ds:**

```
.ds FF "FOOD FIGHT!!"
```

The first field after the **.ds** gives the name of the sting, in this case **FF.** Like a macro name, a string name may be either one or two characters. The second field after the **.ds** gives the definition of the string, in this case

```
"FOOD FIGHT!!"
```

As in this example, you must enclose the definition within quotation marks if it contains spaces.

Be careful whenever you define a macro or a string. If you already have a macro or a string named **X** and you define a new macro or string named **X, nroff** forgets the previous meaning of **X.**

Once you have defined a string, you can refer to it anywhere in your text. The string itself appears in the output text wherever a reference to it appears in the input text. You refer to the string **FF** in the following fashion:

```
\*(FF
```

Use the left parenthesis '(' only when the name of the string is two characters long. If the string name is only a single character, such as **S,** refer to it as follows:

```
\*S
```

As an example, type the following script into **ex7.r**, and process it through **nroff**; do *not* use the **-ms** macro package:

```
.ds FF "FOOD FIGHT!!"
.ds W "WHOOPEE!!"
.ce
From Aristotle's "Poetics"
.br
.sp
A tragedy is the imitation of an action \*(FF
that is serious and also, \*W as having magnitude,
complete in itself, with incidents \*(FF
arousing pity and fear, wherewith to accomplish \*W
\*(FF its purgation of such emotions \*(FF \*(FF.
.bp
```

nroff adjusts the spacings between words in a string but does not hyphenate any word that is within a string. If you use a very short line length, such as two inches, and define a string that includes a three-inch long word, that word would not be hyphenated but would extend past the right-hand margin.

You cannot include a newline character in a string. However, you can spread the definition of a string out over more than one line with the aid of *concealed* newlines (preceded by the backslash character '\'). **nroff** ignores each concealed newline. For example, add the following string to the previous example:

```
.ds PR "GO TEAM \
GO!!!"
```

As you can see, **nroff** ignores concealed newlines anywhere in its input.

Strings Within Strings

You can define a string that has embedded within it a reference to another string. Whenever you refer to the bigger string in your text, **nroff** substitutes the definition of the smaller string for any reference to the smaller string. When you embed strings, though, you should use *two* backslashes to refer to the embedded string, for the same reason that you should use two backslashes to refer to an argument within a macro:

```
.ds S "This string \\*x has embedded \\*y strings"
```

To help understand this better, type following three scripts into your computer and format them with **nroff**. The first script contains proper references to embedded strings (using double backslashes); it works as expected:

```
.ds S "strings \\*X, strings \\*Y, strings \\*Z"
.ds X "here"
.ds Y "there"
.ds Z "everywhere"
\*S
```

The next script contains embedded references that use only single backslashes. Because the embedded strings are defined after the larger string, they are not available when **nroff** defines the larger string, and so the references are ignored:

```
.ds S "strings \*X, strings \*Y, strings \*Z"
.ds X "here"
.ds Y "there"
.ds Z "everywhere"
\*S
```

The third script again contains embedded references using single backslashes. This time, the embedded strings are defined *before* the larger string, and so are available when the larger string is defined:

```
.ds X "here"
.ds Y "there"
.ds Z "everywhere"
.ds S "strings \*X, strings \*Y, strings \*Z"
\*S
```

To avoid unnecessary worry, you should always play it safe and use double backslashes to refer to embedded strings.

Number Registers

You learned in previous sections that **nroff** keeps track of the output page number while it prints its output. You made use of this fact when you created a footer macro that printed page numbers. **nroff** also keeps track of other housekeeping information, such as the current line length, page offset, page length, and vertical position of the last output line. It keeps this information in storage locations called *number registers*.

You can use the *name* of a number register to refer to the number that is stored in it. When you place a reference to a number register in your text, **nroff** substitutes for the name whatever number is currently in the register.

Number register names are one or two characters long, just like macro and string names. You can have a number register with the same name as a string or a macro without confusing **nroff**, even though you cannot give a macro and a string the same name. However, *you* might become confused; **nroff** scripts usually are easier to understand if you keep all macro names, string names, and register names distinct.

Another difference between number registers, macros, and strings is that **nroff** itself does not define any macros or strings (although the **-ms** macro package does), but it does automatically define and update quite a few number registers. You can use these predefined number registers in much the same way that you use registers you define yourself, except that you cannot change their values.

To define a number register, you must specify the *register name* and the *initial value* for the register. The *number register* primitive **.nr** looks like this:

```
.nr X 5
```

Here **X** is the name of the register and **5** is the initial value to store in it. To refer to number register **X** in your text, use **\nX**; if the name is two characters long (for example, **XY**), use **\n(XY**. This is exactly like the way you refer to a string, except that you use the letter 'n' instead of an asterisk '*'. When **nroff** sees a reference to number register **X**, it automatically substitutes the value stored in **X**. As you will see shortly, **nroff** can do arithmetic, and learning to use number registers is an important part of learning to take advantage of **nroff**'s arithmetic abilities.

A reference to a number register can occur anywhere a number would normally occur. For example, if you set register **X** to 5, as above, you can set the line length to five inches as follows:

```
.ll \nXi
```

This command is essentially the same as

```
.ll 5i
```

if the current value of register **X** is 5.

A familiar problem arises when you refer to a number register inside a macro or a string definition. If you use just one backslash, **nroff** substitutes the value in the register for the reference when it first processes the macro or string. If you have not yet defined the number register in your script, **nroff** inserts **0** into the macro or string. Normally, you should use a a double backslash, such as **\\nX** or **\\n(XY**, when referring to a number register within a macro or string. Using the double backslash is particularly important if you *change* the value of the register throughout your script, and want the current value to appear in the macro or string each time you call it.

Try typing the following examples into your computer, and processing them with **nroff**. See if you can describe why **nroff** prints what it does in each case. The first example defines a string with a register reference preceded by a single backslash.

```
.ds S "Here is a number \nX"
.nr X 55
\*S
\nX
```

You should see the following output:

```
Here is number 0
55
```

nroff printed what it did because number register X had not yet been defined when it was called in string S; **nroff** therefore erased the reference to X and substituted zero for it. Number register X was then set to 55, which was printed when the register was specifically called later in the script.

The second example is similar, but now the number register is set *before* the string is called:

```
.nr Y 56
.ds T "Here is a number \nY"
\*T
\nY
```

Now the output is

```
Here is a number 56
56
```

The third example uses a double backslash for the register reference.

```
.ds U "Here is a number \\nZ"
.nr Z 57
\*U
.nr Z 58
\*U
```

This script produces the following:

```
Here is a number 57
Here is a number 58
```

The final example uses a single backslash again.

```
.nr W 59
.ds V "Here is a number \nW"
\*V
.nr W 60
\*V
```

The following is produced:

```
Here is a number 59
Here is a number 59
```

The last example illustrates the danger of using a single backslash to refer to a number register within a string definition. You defined the number register **W** before you defined the string **V**, so the value for **W** was available when **nroff** read the definition of **V**. **nroff** substituted the value when it reads the definition; the reference to the number register **W** is no longer there. You then change the value of **W**, but as you see in the next call of **V**, the change does not affect the number that appears in **V**. In contrast to this, notice in the third example that the double backslash in the definition of **U** allows the reference to number register **Z** to remain within the definition of string **U**. Whenever you change the value of **Z** and then call **U**, **nroff** substitutes the new value of **Z** for the reference to **Z** within **U**.

You can also use the **.nr** primitive to increase or decrease the value in a number register. For example, suppose you initially store the value five in **X**:

```
.nr X 5
```

Incrementing and Decrementing

You can change the value of **X** to 9 by adding 4, as follows:

```
.nr X +4
```

You can then change the value of **X** to 7 by subtracting 2:

```
.nr X -2
```

A plus or minus sign before a number on the **.nr** command line tells **nroff** to add or subtract the given amount to or from the value in the register. Because a negative number is always preceded by a minus sign whereas a positive number usually is not preceded by a plus sign, you can use **.nr** to set a register to a positive value in a way that cannot be imitated for negative values. For example, suppose you again start out with number register **X** set to a value of 5:

```
.nr X 5
```

If you immediately follow this with

```
.nr X 7
```

then **nroff** replaces the value of 5 with 7. The second **.nr** does not increase the value of **X** by 7 to produce 12; rather, it wipes out the previous value of 5 and replaces it by the value 7. The command line to increase **X** by 7 is

```
.nr X +7
```

If you again start with a value of 5 in **X** and want to change the value to -4, you cannot use the following command line:

```
.nr X -4
```

nroff interprets this as a command to *decrease* the current value of **X** by 4, which is not what you intended. This command places the value 1 in **X**, since 5-4=1. If **X** initially has a value of 5 and you want to change the value to -4, you could use the command

```
.nr X -9
```

You can also increase or decrease the value of a number register without using **.nr**. If number register **X** currently has the value 10, the reference **\n+X** increases the value in **X** by 1 to 11 and substitutes the new value for the reference. The value in **X** becomes 11; **nroff** replaces the next reference **\nX** by 11, whereas another reference **\n+X** increments the value in **X** to 12 and replaces the reference by 12. Similarly, if number register **XY** currently has the value 15, the reference **\n+(XY** increases the value in **XY** to 16 and replaces the reference by 16.

You can also decrease a register's value. The reference **\n-X** decreases the current value in **X** by 1 and substitutes the new value for the reference. Likewise, the reference **\n-(XY** decreases the current value in **XY** by 1 and substitutes the new value for the reference.

You can change the size of the increment or decrement by means of another option to the **nr** command. If you define **X** with

```
.nr X 1 5
```

then **nroff** sets the value of **X** to 1 and sets the increment value for **X** to 5. The next reference **\n+X** increments the value in **X** from 1 to 6 (the '+' now causes **nroff** to add 5 to the current value of **X** rather than adding 1) and substitutes 6 for the reference. In the same manner, **\n-X** subtracts 5 from the current value of **X** and substitutes the new value for the reference. This is convenient if you plan to repeatedly increment or decrement **X** by the same fixed amount. If you wish to change the size of the increment, simply redefine **X** with another **.nr** that specifies the new initial and increment values. If you define a number register but do not specify an increment value, **nroff** assumes the increment value to be 1.

The following example of a macro illustrates a typical use of a number register and incrementing.

```
.nr W 1                                  \" set W to 1, inc by 1
.ds X "Here's Wrestler No. \\nW,"        \" define string X
.de B                                    \" define macro B
.br
\\*X \\$1!!!                             \" define arg to macro B
.nr b \\n+W                              \" increment W
..                                       \" end definition
.B "Alex 'Killer' Bovine"                \" call B with arguments
.B "William 'Crusher' Risible"
.B "Vlad 'the Impaler' Acephalous"
.bp                                      \" force printing of page
```

to produce the following output:

```
Here's Wrestler No. 1, Alex 'Killer' Bovine!!!
Here's Wrestler No. 2, William 'Crusher' Risible!!!
Here's Wrestler No. 3, Vlad 'the Impaler' Acephalous!!!
```

A reference to a number register may appear any place a number can normally appear. For example:

```
.nr X \nY \nZ
```

sets register **X** to the value of register **Y** and sets the increment for **X** to the value of register **Z.**

As mentioned before, **nroff** performs arithmetic. It understands and evaluates properly formed arithmetic expressions involving numbers, references to number registers, the arithmetic operators '+', '-', '*', '/', '%', and parentheses. The first four operators represent addition, subtraction, multiplication, and division. The '%' is the *modulus* or *remainder* operator: the value of 7%3 is 1, which is the remainder when 7 is divided by 3.

One word of caution: **nroff** evaluates expressions from left to right without any preference for performing some operations before others. For example,

```
.nr X 5+4*3/9
```

stores 3 in **X**. **nroff** does not perform the multiplication and division before the addition, as you might expect.

Another important fact is that number registers hold only integers. If you write

```
.nr X 3.6
```

nroff truncates the value 3.6 and stores 3 in **X**. Also, an assignment such as

```
.nr X 3.9*3.9
```

stores 9 in **X**; **nroff** truncates each factor before it performs the multiplication. The assignment

```
.nr X 0.4*8
```

stores 0 in **X** rather than 3: truncation occurs before **nroff** performs the multiplication rather than after.

A final word of caution: when you use numbers with commands other than **.nr**, the results may *not* be what you expect. **nroff** understands several different units of measurement and converts between units automatically. The next section explains units and conversion in detail.

Units of Measurement

As mentioned above, **nroff** maintains many number registers during processing. For example, it stores the current page length in the register **.l** (Note that the period '.' is actually part of the name of this register.) If you set the line length to five inches with the command

```
.ll 5i
```

nroff stores the length in register **.l** automatically; however, if you print the value in register **.l** by entering **\n(.l**, you find the value is 600. What does this mean?

Many **nroff** commands require that you specify lengths or measurements as arguments. You are already familiar with many of these commands: for example, **.ll**, **.po**, **.pl**, and **.lt**. **nroff** accepts various units of measurement, but for purposes of calculation, it converts each into a basic unit called a *machine unit*, which is abbreviated **u**. A machine unit is 1/120 of an inch long. Because one inch is 120 machine units, the length of a five-inch line is 5 times 120, or 600 machine units.

The conversion table for units of measurement is as follows:

inch:	1**i** = 120**u**
vertical line space:	1**v** = 20**u**
centimeter:	1**c** = 47**u**
em:	1**m** = 12**u**
en:	1**n** = 6**u**
pica:	1**P** = 20**u**
point:	1**p** = 1**u**

Most of these are traditional typesetting terms.

As noted briefly earlier, **nroff**'s output actually consists of a sequence of characters. It is useful, though, to think of the output as being "printed" at ten characters per inch (Pica or 10-pitch spacing) and six lines per inch. Many output devices use this spacing. With these assumptions, **5i** is equivalent to five inches of printed output.

Every **nroff** command has a default unit of measurement. For example, the default unit for **.ll** is **m**, whereas the default unit for **.sp** is **v**. If you type

```
.ll 5
```

nroff interprets it not as five inches or five centimeters, but as 5**m**, which it converts to 5 times 12, or 60 machine units (60**u**).

nroff always assumes a unit specification as part of each number and automatically converts each number and its unit specification into machine units. If you append an explicit unit specification to the number, **nroff** uses it; if you do not, **nroff** uses the default unit for the command.

For example, suppose you write the following commands:

```
.nr X 2i
.ll \nX
```

What line length results? The first command stores the number 2 times 120, or 240, in register **X**. The second command is therefore equivalent to typing

```
.ll 240
```

However, the default unit for **.ll** is **m**. Because 1**m** equals 12**u**, **nroff** sets the line length to 12 times 240, or 2,880 machine units. If you wanted a line length of two inches to result from the above commands, you will be unpleasantly surprised, because 2**i** equals only 240**u**. Instead, you should write:

```
.nr X 2i
.ll \nXu
```

By including the **u** in the **.ll** primitive, you do not accidentally multiply your results by 12, as happened earlier.

You should think of the unit specification as a part of a number. Because **nroff** accepts so many different units of measurement, a number without a unit specification is ambiguous. What does '5' mean? Five inches? Centimeters? Ems? **nroff** must know what unit of measurement you are using. If you think of the unit specification as a part of the number, you will have less trouble with potentially mystifying situations like the following. As mentioned, number registers store only integers and **nroff** truncates each number in an arithmetic expression to an integer before evaluating the expression. Therefore, the following stores 0 in register **X**:

```
.nr X 0.4*9
```

But now try the following:

```
.nr X 0.4i
\nX
```

This does not store 0 in **X** like the previous command; it stores 0.4 times 120, or 48 in **X**. The 0.4 is not truncated to 0 here! Truncation occurs *after* conversion to machine units, so **nroff** truncates 0.4**u** in the first example. But the number in the second example is given in inches **i** instead of machine units **u**. **nroff** converts it to **u** before truncating to get an integer.

As another example, the following stores 1 in **X**:

```
.nr X 0.01i
```

nroff converts 0.01 inches to 0.01 times 120, or 1.2**u**, and then truncates 1.2 to 1.

The following command illustrates that **nroff** understands *each* number in an arithmetic expression to have an attached unit specification, whether you supply one or not.

```
.ll 2*8
```

Recall that **nroff** stores the current line length in the register **.l**; if you type

```
\n(.l
```

you will receive the number 2,304. **nroff** interprets the 2 as 2**m** and the 8 as 8**m**, because the default unit for **.ll** is **m**. Then it converts each to machine units and multiplies to give the result: (2*12)*(8*12), or 2,304.

Consider one final example that illustrates the unusual consequences of seemingly innocent assignments. Suppose you set the page offset as follows:

```
.po 8/3
```

nroff stores the current page offset in register **.o**. To see what number it stores there, type

```
\n(.o
```

You see that the page offset is 2. Because the default unit for **.po** is **m**, the calculation is (8*12)/(3*12)=8/3, which **nroff** truncates to 2. Two machine units is equivalent to only 1/60 of an inch. This is not a physically reasonable value for most typewriter-like devices, so a page offset of 0 characters results. On the other hand,

```
.po 8/3u
```

produces a page offset of approximately 1/4 of an inch.

Conditional Input

Now that you have been introduced to number registers, you can use them in conjunction with powerful *conditional commands* to create more elaborate **nroff** scripts.

To see how conditional statements help you construct an **nroff** script, consider again the problem of creating header and footer macros. Earlier, you constructed macros that skipped space at the top of the page and printed the page number at the bottom of each page.

Suppose, however, that you are formatting a paper that has a title. You want to print the page number for page 1 at the bottom of the page, and to print the rest of the page numbers at the top of the page. Both the header and the footer need some kind of conditional mechanism to perform differently on the first page than on subsequent pages. On page 1, the header should skip to where the title will be printed; on other pages, the header should print the page number. On page 1, the footer should print the page number; on other pages, the footer should leave a block of blank space at the bottom of the page.

To execute commands conditionally, use the *if/else* commands **.ie** and **.el**, which are demonstrated in the following example. Note that the formation **''**, which is used with the **.tl** command, represents two apostrophes, *not* a quotation mark.

```
.de HD          \" define header
.ie \\n%=1 .A
.el .B          \" else do B
..
.de A           \" define macro A
.sp |1.0i       \" space down to 1 inches from top of page
..
.de B           \" define macro B
'sp 2v          \" skip 2 spaces
.tl ''- % -''   \" print page no.
'sp |1.0i       \" skip to 1 inch from top of page
..
.de FO          \" define footer
.ie \\n%=1 .C   \" if page no. is 1 then do C
.el .D          \" else do D
..
.de C           \" define macro C
'sp |-4v        \" move to 4 in. above bottom of page
.tl ''- % -''   \" print page no.
'bp             \" begin new page
..
.de D           \" define macro D
'bp             \" begin new page
..
```

As you can see, the **.ie** and **.el** commands always occur in pairs. **.ie** consists of three parts: the command name **.ie**, then a *condition* that **nroff** tests, followed by a *command* **nroff** performs if the condition is true. If the condition on the **.ie** command line is not true, **nroff** performs the command on the **.el** line instead.

In the example, each conditional invokes a macro on the command line. Actually, the conditional can specify *input* text rather than the command after the condition. If you want to execute several commands or include several text lines conditionally, enclose the lines with the special sequences '\{' and '\}'.

Note, too, that one other new element was introduced in the construction of these macros. Some of the **.sp** commands have a vertical bar immediately in front of the measurement; for example,

```
.sp |1.0i
```

Normally, when **nroff** sees a command like **.sp 1.0i**, it moves down one inch on the output page. The movement is relative to where **nroff** happens to be on the output page when it received the request. The vertical bar tells **nroff** that the following measurement is an *absolute* measurement, measuring either from the top of the page (if positive) or from the bottom of the page (if negative). Therefore,

```
.sp |1.0i
```

tells **nroff** to move to one inch from the top of the page;

```
.sp |(-4v)
```

tells it to move to four vertical spaces from the bottom of the page.

The **.if** primitive is formed exactly like **.ie**. Unlike **.ie**, which must always be used with **.el**, the **.if** command may be used by itself. If the condition on the **.if** line is true, **nroff** performs the command that follows the condition; if the condition is false, it ignores the command altogether.

This chapter ends with two substantial examples that incorporate most of what you have studied so far. To illustrate the use of conditionals, the first example begins each even paragraph of output with the phrase **Even Paragraph:** and begins each odd paragraph with the phrase **Odd Paragraph:**. Type this into the file **ex8.r**, and process it through **nroff** without using the **-ms** macro package, and as before, there is no need to copy the comments:

```
.wh 0 HD          \" set header trap
.wh -2i FO        \" set footer trap
.nr EO 1          \" set EO register to 1
.po 2i            \" page offset 2 inches
.pl 6i            \" page length 6 inches
.lt 4i            \" title length 4 inches
.ll 4i            \" line length 4 inches
.de HD            \" define header
'sp |(1i-1v)      \" space down to 1 inch minus 1 line
.tl ''\\*(WS''    \" set WS macro in title
'sp |1.5i         \" space down to 1.5 inches
..
.de FO            \" define footer
'sp |(3i+3v)      \" space down to 3 inches plus 3 lines
.tl ''- % -''     \" set page number in footer
'bp               \" begin new page
..
.ds WS "From the Devil\'s Dictionary"
.                 \" define string WS
.de PP            \" define paragraph macro
.ie \\n(EO=0 .EP \" if EO = 0 (even) then do EP
.el .OP           \" else do OP
..
.de EP            \" define EP (even paragraph)
.br
.nr EO 1          \" set register EO to 1
.sp 1v            \" skip 1 line
.ll 4i            \" set line length to 4 inches
.lt 4i            \" set title length to 4 inches
\\*E              \" insert string E
..
.ds E "Even Paragraph:"
.                 \" define string E
.de OP            \" define macro OP (odd paragraph)
.br
.nr EO 0          \" set register EO to 0
.sp 1v
.ll 3i            \" set line length to 3 inches
.lt 3i            \" set title length to 3 inches
\\*O              \" insert string O
..
.ds O "Odd Paragraph:"
.                 \" define string O
```

```
.PP
Debt, n. An ingenious substitute for the whip
and chain of the slave-driver.
.PP
Bore, n. One who talks when you wish him to listen.
.PP
Brandy, n. A cordial composed of one part
thunder-and lightning, one part remorse, two parts
bloody murder, one part death-hell-and-the-grave,
and four parts clarified Satan.
.PP
Responsibility, n. A detachable burden easily
shifted onto the shoulders of God, Fate, Fortune,
Luck, or one's neighbor.
```

This example uses an "even/odd" register called **EO** to determine whether you are beginning an even or an odd paragraph. To distinguish between even and odd paragraphs, it uses a line length of four inches for even paragraphs and one of of three inches for odd paragraphs. It changes the title length with each paragraph, so **nroff** centers the page number with respect to whichever kind of paragraph happens to occur at the bottom of a page.

The final example illustrates a loop constructed with the **if/else** commands. The first paragraph is six inches long with no page offset; each succeeding paragraph is one inch shorter with a page offset one inch longer. The line length of the sixth paragraph is one inch; the next paragraph renews the cycle with a six-inch line length. Type this into file **ex9.r**, and process it as you did the above example:

```
.nr PO 0 1        \" set register PO to 0, increment by 1
.de PP            \" define paragraph macro
.ie \\n(PO=6 .A   \" if register PO=6 then do A
.el .B            \" else do B
..
.de A             \" define macro A
.br
.nr PO 0          \" set register PO to 0
.nr LL 6-\\n(PO   \" set register LL to 6 minus PO
.ll \\n(LLi       \" set line length to LL inches
.po \\n(POi       \" set page offset to PO inches
.nr PO \\n+(PO    \" increment register PO
.sp               \" skip a space
..
.de B             \" define macro B
.br
.nr LL 6-\\n(PO   \" set LL to 6 minus PO
.ll \\n(LLi       \" set line length to LL inches
.po \\n(POi       \" set page offset to PO inches
.nr PO \\n+(PO    \" increment register PO
.sp               \" skip a space
..
.PP
Future, n. That period of time in which our affairs prosper,
our friends are true, and our happiness is assured.
.PP
Gallows, n. A stage for the performance of miracle plays, in
which the leading actor is translated into heaven.
.PP
```

```
Geneaology, n. An account of one's descent from an ancestor
who did not particularly care to trace his own.
.PP
Guillotine, n. A machine which makes a Frenchman shrug
his shoulders with good reason.
.PP
History, n. An account most false, of events
most unimportant, which are brought about by
rulers mostly knaves, and soldiers mostly fools.
.PP
Idiot, n. A member of a large and powerful tribe
whose influence in human affairs has always been
dominant and controlling.
.PP
Kiss, n. A word invented by the poets as a rhyme
for "bliss".
```

You should try this example to see verify that "loop" works as advertised.

Environments and Diversions

Another aspect of **nroff**'s power is the ability to shift from one *environment* to another.

The **nroff** *environment* is the overall manner in which **nroff** processes your input text. The environment's definition includes such aspects as line length, fill and adjust modes, and indentation.

nroff allows you to define three independent environments, called **0**, **1**, and **2**. In each, you can set as you wish such parameters as line length, filling, adjustment, and indentation. You can call a different environment with the **.ev** command; the parameters you define for the new environment control text processing until you change them within the present environment or shift to another environment.

Not all **nroff** parameters change when you switch to a new environment. For example, different environments do *not* have independent page offsets; the **.po** command affects all environments. Parameters that may be set to different values in different environments are *environmental parameters*; parameters that cannot be switched according to environment, like page offset, are *global parameters*. Macro and string definitions are global.

When you first call **nroff**, you are by default in environment 0. In all the examples used in this tutorial thus far, everything happened in environment 0. The following example illustrates how to switch back and forth between environments. Type in the following **ex10.r** and process it to see the output as you go along.

```
.po 1i          \" set global page offset to 1 inch
.ll 4i          \" set line length in ev 0 to 4 inches
.de PP          \" define paragraph macro
.sp
.ti 0.5i        \" indent first line 1/2 inch
..
.PP
The heart of the righteous studieth to answer,
but the mouth of the wicked poureth out evil things.
.br
.ev 1           \" switch to environment 1
.ll 3i          \" set line length to 3 inches
.ls 2           \" line spacing now double space
.PP
A froward man soweth strife, and a whisperer
separateth chief friends.
.br
.ev             \" return to previous ev (0)
.PP
It is naught, it is naught, sayeth the buyer;
but when he is gone his way, then he boasteth.
.br
.ev 1           \" switch to ev 1
.PP
Wealth maketh many friends; but the poor is separated
from his neighbors.
.br
.ev             \" return to ev 0
```

The first **.ll** command sets a line length of four inches in environment 0. After defining the paragraph macro **.PP** and an initial paragraph in environment 0, you switched to environment 1 with the command

```
.ev 1
```

You now enter a new environment. If you do not explicitly set environmental parameters, such as line length, **nroff** automatically uses default values for them. **nroff** assigns the same default values in environments 1 and 2 as it does in environment 0.

The line length in environment 1 is set to three inches with the output text double-spaced. The *line space* primitive

```
.ls 2
```

leaves one blank line between each output line. Thus, paragraphs processed in environment 0 have four-inch single-spaced lines, whereas paragraphs processed in environment 1 have three-inch double-spaced lines.

The example used the command line

```
.ev
```

without an argument to *leave* environment 1. This leaves environment 1 and restores (or "pops") the previous environment — in this case, environment 0. The next time you enter

environment 1, you will not need to set the line length to three inches again: the value stays in effect in environment 1 until you specifically change it. The same is true of all environmental parameters.

To understand how **nroff** switches between environments, imagine that you have a set of plates, each marked with either a **0**, a **1**, or a **2**. You have as many plates of each type as you wish. You stack the plates on a table; the top plate represents your current environment. You begin with a '0' plate on the table, to represent the initial environment when you enter **nroff**.

Switching to environment 1 with the command **.ev 1** corresponds to placing a '1' plate on top of the '0' plate. You can again change the stack of two plates either by placing a new plate on top of the stack, or by removing the top plate from the stack: the former corresponds to calling a new environment, whereas the latter corresponds to restoring the previous environment with the command line **.ev**.

Because you can have as many plates of each type as you wish, you can call environment 1, then call environment 2, then restore environment 1, then call environment 0, and so on. The command **.ev** *N*, where *N* is 0, 1, or 2, places (or "pushes") a plate onto the stack; the command **.ev** removes (or "pops") the top plate from the stack.

To illustrate this, add the following text to the end of the previous example. Use a piece of paper and pencil to keep track of how the **.ev** commands add or remove environments. Because the line lengths are different in each environment, it should be easy to tell in which environment **nroff** has processed each paragraph:

```
.ev 2            \" introduce environment 2
.ll 5i           \" set line length
.in 1i           \" set indentation
.PP              \" paragraph in ev 2
A poor man that oppresseth the poor is like
a sweeping rain which leaveth no food.
.br
.ev 0            \" push ev 0
.PP
As a roaring lion, and a ranging bear; so is
a wicked ruler over the poor people.
.br
.ev 1            \" push ev 1
.PP
Wrath is cruel, and anger is outrageous;
but who is able to stand before envy?
.br
.ev 2            \" push ev 2
.PP
A good name is rather to be chosen than
great riches; and loving favour rather than
silver and gold.
.br
.ev 0            \" push ev 0
.PP
Pride goeth before destruction, and an haughty
spirit before a fall.
.br
.ev              \" return to ev 2
.ev              \" return to ev 1
.PP
He that answereth a matter before he heareth it,
it is folly and shame unto him.
.br
.ev              \" return to ev 0
.ev              \" return to ev 2
.PP
A merry heart doeth good like a medicine, but a
broken spirit drieth the bones.
.br
```

Buffers

Earlier, it was shown that **nroff** uses a buffer to assemble words from its input into output lines. Actually, each environment has its own buffer. Switching to a new environment does *not* cause a break. Suppose you are currently in environment 1 with an unfinished line in the buffer. When you give the command **.ev 2**, the unfinished line remains undisturbed in the environment 1 buffer until you return to environment 1. Text you process in the meantime in environment 2 or in environment 0 has no effect on the partial line in the

environment 1 buffer, because **nroff** assembles text processed in other environments in different buffers.

In the following example, you process some text in environment 0 and then switch to environment 2. Any partial line collected in environment 0 when you switch to environment 2 waits patiently in the buffer until you return to environment 0 and issue the break command to flush the buffer. You then return to environment 2 and flush any partially filled line left when you restored environment 0. Enter the following into the file **ex11.r** and process it through **nroff**:

```
.ll 3i          \" set line length in ev 0
.po 2i          \" set page offset in ev 0
This is environment 0.
.ev 2           \" introduce ev 2
This is environment 2
.br             \" flush ev 2 buffer
.ev             \" return to ev 0
.br             \" flush ev 0 buffer
```

As you can see, the order of the two sentences is reversed from the way you entered them. If you were to delete the **.br** commands after the texts in **ex10.r**, the output would be very badly affected.

Headers and Footers

A common use of environment switching is for the creation of header and footer macros. As the following example demonstrates, the length of title set by the **.lt** command is an environmental parameter. The following constructs header and footer macros that print strings of asterisks in the margins above and below the text; type it into your computer as **ex12.r**:

```
.wh 0 HD          \" set header trap
.wh |2.5i FO      \" set footer trap
.de HD            \" define header macro
.ev 1             \" define ev 1
.lt 5i            \" set title length to 5 inches
'sp 3v            \" move down three spaces
.tl '****''****' \" define header title
'sp 2v            \" skip two more spaces
.ev               \" pop environment
..
.de FO            \" define footer macro
'sp 2
.ev 1             \" push ev 1
.tl '****'%'****'\" define footer title
.ev               \" pop environment
'bp               \" begin new page
..
.ll 4i            \" set line length in ev 0
.pl 3i            \" set page length
.in 1i            \" set indentation
.po 2i            \" set page offset
.de PP            \" define paragraph macro
.sp 1
.ti 0.5i          \" indent 1st line 1/2 inch
..
.PP
When in the course of human events ...
```

The following section explains why header and footer macros often use a different environment.

More About Fonts

As earlier described in some detail, **nroff** output includes representations for **boldface** and *italic* characters, in addition to the normal Roman characters. The visual appearance of boldface and italic characters depends on the device you use to print your **nroff** output.

If you want a word or a phrase to appear in boldface, enclose the word or phrase between **\fB** and **\fR**:

```
The last word of this sentence appears in \fBboldface\fR.
```

The sequence **\fB** tells **nroff** to print in boldface, whereas the sequence **\fR** tells **nroff** to return to the Roman font. Italics are used in a similar manner:

```
An entire phrase \fIappears in italics\fR.
```

To print more than a few words in a different font, you should use the *font* command **.ft**:

```
.ft I
Here is text you want to
appear in italics.
.ft R
```

The initial **.ft I** switches the print to italic font, and the concluding **.ft R** returns it to Roman font. As you might have suspected, the command **.ft B** switches to boldface.

You have two additional options when you use the **.ft** primitive. The command **.ft P** returns to the *previous* font. You can use **.ft P** within a macro or a string to return to the previous output font, even though you do not know which font was previously in effect. You can also use the sequence **\fP** to return to the previous font. The **.ft** primitive without an argument tells **nroff** to return to the Roman font.

In scripts that frequently change fonts, you should switch to a new environment for header and footer macros, in order to protect their font settings.

Diversions

The *diversion* is a powerful feature that allows you to suspend outputting lines until **nroff** has collected all of a block of text. For example, suppose you use **nroff** to format a chapter of a book. The chapter includes footnotes at various places in the text; you want **nroff** to collect these footnotes and print at the end of the chapter. To do this, **nroff** must store the processed footnote text somewhere until the end of the chapter, when you want it printed. Where do you store the text until the time comes for it to appear? To handle situations like this, **nroff** provides a *diversion* mechanism: you can *divert* text into temporary storage within a macro.

Diversion normally involves passing to a new environment to process the footnote without causing a break in the main environment. When the text of the diversion ends, **nroff** returns to the main environment, again without causing a break, and continues processing just as if the text of the note had never been in the input.

However, before you attempt to write a footnote macro, type the following text into the file **ex13.r**, and process it with **nroff**. This example illustrates the basic features of diversion. The example exchanges two paragraphs of text, so that **nroff** prints the second before the first.

```
.di X           \" divert the following to macro X
.sp
A soft answer turneth away wrath:
but grievous words stir up anger.
.br             \" send last line of paragraph to X
.di             \" end diversion
.sp
He that is slow to anger is better than the
mighty; and he that ruleth his spirit than he
that taketh a city.
.br
.sp
.X              \" print the paragraph diverted to X
```

The new command here is the *divert* primitive **.di**. The command **.di X** tells **nroff** to divert the text that follows into macro **X**; the matching **.di** with no argument marks the end of the diversion.

The break is necessary before the end of the diversion because **nroff** diverts *processed* text into the macro. Without the break, **nroff** would not divert any partially filled line in its buffer to **X**; the last few words of diverted text might not form a complete line in the buffer, so **nroff** might not divert them. However, if you break the input before you end the diversion, **nroff** will also divert those last few words.

As you saw earlier, the **.br** command must be used to flush that environment's buffer before switching environments.

The next example, **ex14.r**, illustrates a similar point.

```
.br              \" clear buffer
testword         \" put 'testword' into buffer
.di X            \" divert to X
Piracy, n. Commerce without its folly-swaddles,
just as God made it.
.br              \" divert last line
.di              \" end diversion
.X               \" print text in X
```

Here **nroff** diverts **testword** into **X** along with the text between **.di X** and **.di**. Why did this happen? The command **.di X** does *not* cause a break. Because you did not pass to a new environment in this example before you diverted, **nroff** formed the diversion text in the same buffer in which it stored **testword**. You did not break the input, so **nroff** appended the diverted text to **testword**.

To make sure **nroff** diverts only text between **.di X** and **.di** into **X**, do one of the following: If you want to process the diverted text within the current environment, empty the buffer by inserting the **.br** command before you start the diversion. If you switch to a new environment before starting the diversion, flush the buffer for the new environment before you begin to process diverted text.

Diverting processed text into a macro that already holds material will erase whatever had already been stored there. In some cases, such as the footnote example, you need to append information into the same macro. The *divert and append* variation **.da** of the diversion construction allows you to do so. The following example, **ex15.r**, demonstrates this command:

```
.ll 3i          \" set line length
.po 2i          \" set page offset
.de PP          \" define paragraph macro
.br
.sp 1
.ti 0.5i        \" indent first line 1/2 inch
..
.di X           \" divert the following into X
.PP
Litigation, n. A machine which you go into as a pig
and come out of as a sausage.
.br
.di             \" end diversion
.X              \" print what is in X
.br
.da X           \" divert and append material into X
.PP
Inventor, n. A person who makes an ingenious arrangement
of wheels, levers and springs, and believes it
civilization.
.br
.di             \" end diversion
.X              \" print what is now in X
```

In this example, you first diverted a single paragraph into the macro **X**. **nroff** stored in **X** the *processed* paragraph; in other words, the command line **.PP** is *not* stored in **X**; its *output* is. When you invoke **X** with the command line **.X**, **nroff** takes the processed text in **X** as input. To **nroff**, there is no difference between processed text and unprocessed text as input: it processes the contents of **X** in the current environment, just like any other text. Therefore, **nroff** processes diverted text *twice*: first when it stores the text within the macro, and again when you invoke the macro.

The fact that **nroff** processes diverted text twice can cause problems if you are not careful. Fortunately, nothing strange happens in the example above. You store a processed paragraph with lines three inches long in **X**. When you invoke **X**, the line length is three inches. Because each line in **X** is already exactly three inches long, nothing happens to it when reprocessed; the layout of the output paragraph is unchanged.

But now, consider what happens in the following example, **ex16.r**:

```
.ll 3i          \" set line length
.po 2i          \" set page offset
.de PP          \" define PP macro
.sp 1
.ti 0.5i        \" indent first line 1/2 inch
..
.di X           \" divert following into X
.ev 2           \" push environment 2
.ll 4i          \" set line length to 4 inches
.PP
Justice, n. A commodity which in a more or less
adulterated condition, the State sells to the
citizen as a reward for his allegiance, taxes
and personal service.
.br
.ev             \" pop environment (return to ev 0)
.di             \" end diversion
.X
```

A pargraph processed in environment 0 in this example has three-inch lines; you want your diverted paragraph to have four-inch lines. However, when you print the diverted paragraph with the command line **.X**, what happened? **nroff** did *not* print four-inch lines. The four-line text lines set in environment 2 were reprocessed into three-inch lines when the diversion macro is called in environment 0.

There are two ways to prevent such disasters. First, if you wish to invoke **X** in the main environment, use no-fill mode:

```
.nf             \" begin no-fill mode
.X
.fi             \" return to fill mode
```

In no-fill mode, **nroff** outputs lines of input exactly as it receives them, so it keeps four-inch lines four inches long and does not change the format of the diverted text. The second strategy is to return to environment 2 and then invoke **X**; again, the format of the diverted paragraph does not change, because the line length in environment 2 is four inches.

```
.ev 2           \" push environment 2
.X
.ev             \" restore original environment
```

A Footnote Macro

The footnote macro that follows does not print notes at the bottom of each page; rather, it prints everything at the end of the chapter. In the processed text, number register **Fn** is used to keep track of the footnote number; the footnote number will be printed in square brackets where the footnote originally appeared in the text.

Type this macro into the file **ex17.r**. If you wish to use it in your text processing, transfer it to the directory **/usr/lib** under the name **tmac.fn**. Then, whenever you wish to use this macro, be sure to include the option

```
-mfn
```

when you invoke **nroff**:

```
.de FN              \" define macro FN
[\\n+(Fn]           \" print footnote no. in main text
.ev 1               \" push environment 1
.da Z               \" divert and append following into Z
.sp
\\n(Fn. \\$2, \\fI\\$1\\fR,
    \\$3, \\$4.     \" format & print footnote in Z
.br                 \" flush diversion buffer
.di                 \" end diversion
.ev                 \" pop environment (return to ev 0)
..
```

Note that requests to change fonts are preceded by double backslashes, because they are within a macro. The change to the italic font prints the first macro argument, which should be the title of the work, in italics. Number register **Fn** contains the number of the last footnote; you should initialize it with the command

```
.nr Fn 0 1
```

As shown above, each footnote entry in your text should have four arguments. In your input text, each footnote will look like this:

```
.FN "Personal narrative of a pilgrimage to\
El-Medinah and Mecca" "Richard F. Burton"\
London 1856.
```

When you print the diversion **.Z** at the end of the chapter, each footnote will be laid out as follows:

8. Richard F. Burton,
Personal narrative of a pilgrimage to
El_Medinah and Mecca,
London, 1856.

Command Line Options

In the previous sections, you learned how to control **nroff** by including *commands* in the input along with the *text*. You can also supply information in another way: on the command line you type to call **nroff**. Unlike the commands discussed above, this information is *not* part of the script you input into **nroff**.

You already know about some simple **nroff** command lines. For example, the command

```
nroff
```

forces **nroff** to accept input from the keyboard (sometimes called the *standard input)* and print output on the terminal (the *standard output)*. Type **<ctrl-D>** (that is, hold down the **ctrl** key and type **D**) to exit from **nroff** if it is reading input from your terminal.

The command line

```
nroff script1.r
```

forces **nroff** to take accept input from the file **script1.r** instead of from your terminal, while the command

```
nroff -ms script.r
```

processes **script1.r** with the **-ms** macro package. You can also redirect **nroff** output to another file **target:**

```
nroff -ms script1.r >target
```

The general form of the **nroff** command line is:

nroff *[option ...] [file ...]*

This means that the command line consists of the **nroff** command, followed by zero or more *options*, followed by zero or more *files*. **nroff** processes each named *file* and prints the result on the standard output (the terminal, unless redirected). If no *file* argument is given, as in the first example above, **nroff** reads from the standard input.

Each *option* on the command line must begin with a hyphen '-' to distinguish it from a *file* specification. Using **nroff** with the **-ms** macro package is one example of entering an option. In general, the **-m** option takes the form

-m*name*

which means the option consists of the characters **-m** immediately followed by a *name*. This tells **nroff** to process the macro package found in the COHERENT file

/usr/lib/tmac.*name*

For example, the **ms** macro package discussed in chapter 2 is in the file **/usr/lib/tmac.s**, whereashe **man** macro package used for the **man** command and to process manual pages is in the file **/usr/lib/tmac.an.**

Any macro packages that you customize for your own use should be stored in the directory **/usr/lib** under such a name if you wish to use them with the **-m***name* option.

The **-i** option tells **nroff** to read input from the standard input after processing each given *file*. This allows you to supply additional input interactively from your terminal.

The **-x** option tells **nroff** not to move to the bottom of the last output page when done. This is especially useful if you want to see the output on the screen of a CRT terminal.

The **-n***N* option sets the page number of the first output page to the number *N*, rather than starting at page 1. This is useful for processing large documents with input text in several files which **nroff** processes separately.

The **-r***XN* option sets the value of number register *X* to *N*. This option lets you initialize number registers when you invoke **nroff**.

The COHERENT system provides many useful features which can be helpful while you are using **nroff**. In particular, you can use a number of special characters. The *stop-output* and *start-output* characters, usually **<ctrl-S>** and **<ctrl-Q>**, stop and restart output on your terminal. The *interrupt* character, usually **<ctrl-C>**, interrupts program execution; you can use it to stop an **nroff** command if you typed the command line incorrectly. The *kill* character, usually **<ctrl-\>**, also terminates program execution. Some COHERENT systems use different characters than those mentioned above; consult *Using the COHERENT System* for details.

For Further Information

The Lexicon entry for **nroff** summarizes its primitives, dedicated number registers, escape sequences, and command-line options. The related program **troff** also performs text formatting, except that it produces proportionally spaced output that can be printed on printers that support the Hewlett-Packard Page Control Language (including the LaserJet and DeskJet families of printers) or on printers that support the PostScript page-control language. See the Lexicon entry for **troff** for details on how to use this program.

The Lexicon also has entries for two macros packages that are included with the COHERENT system: **man**, which produces manual pages similar to those that appear in the Lexicon; and **ms**, which performs formatting somewhat similar to that seen in this tutorial. You will find that these two packages already perform practically all of the formatting tasks that you will commonly need to do.

The error messages generated by **nroff** are given in the appendix at the rear of this manual.

Section 12:

Introduction to the sed Stream Editor

This is a tutorial for the COHERENT editor **sed**. It describes in elementary terms what **sed** does.

This guide is meant for two types of reader: the one who wants a tutorial introduction to **sed**, and the one who wants to use specific sections as references.

Related tutorials include *Using the COHERENT System*, which presents the basics of using COHERENT and introduces many useful programs, and the tutorials for the interactive line editor **ed** and for the screen editor MicroEMACS.

In a nutshell, **sed** edits files non-interactively; that is, **sed** applies your set of commands to every line of the file being edited. Although **sed** is not as easy to control as **ed** or MicroEMACS, both of which are interactive, it can edit a large file very quickly. You can use **sed** to change computer programs, natural language manuscripts, command files, electronic mail messages, or any other type of text file.

Getting to Know sed

sed is a text editor. It reads a text file one line at a time, and applies your set of editing commands to each line as it is read. Because it does not ask you for instructions after it executes each command, **sed** is a *non-interactive* text editor.

The advantages of **sed** are that it can readily apply the same editing commands to many files; it can edit a large file quickly; and it can readily be used with *pipes*. A pipe takes the product of one program and feeds it into another program for further processing. If you are unsure of how a pipe works, refer to *sh Shell Command Language Tutorial*.

sed resembles closely **ed**. **sed** and **ed** use almost all of the same commands, and locate lines in much the same way. However, there are important differences between **ed** and **sed**. **ed** is interactive: when you give **ed** a command from the keyboard, it executes that command immediately and then waits for you to enter the next command. **sed**, on the other hand, accepts your editing commands all at once, either from the keyboard or, more often, from a file you prepare; then, as it reads your text file one line at a time, it applies every command to every line of text. Therefore, *addressing* (that is, telling the program what commands should be applied to which lines) is much more important with **sed** than with **ed**.

Keep in mind, too, that **sed** does not change your original text file; rather, **sed** copies it, changes it, and sends the edited file either to the standard output or to another file that you name in the command line.

Getting Started

Here are a few exercises to introduce you to **sed**. Type them into your COHERENT system to get a feel for how **sed** works.

As explained above, **sed** applies a set of editing commands to your text file. To edit a file with **sed**, you must prepare three elements: (1) the text file that you wish to edit; (2) a command file (or *script*) that contains the **sed** commands you want to apply to the text file;

and (3) a command line that tells the COHERENT system what you want done and with which files.

To begin, then, type the following text into your computer using the **cat** command. (Remember that **<ctrl-D>** is typed by holding down the *ctrl* key and simultaneously typing *D*.)

```
cat >exercise1
No man will be a sailor who has contrivance enough
to get himself into a gaol; for being in a ship is
being in a gaol, with the chance of being drowned.
<ctrl-D>
```

Now, type in the following **sed** script. This script will substitute *jail* for *gaol*:

```
cat >script1
s/gaol/jail/g
<ctrl-D>
```

The last step is to prepare the command line. The command line consists of the **sed** command, the options that tell **sed** where its instructions will be coming from (either from a file or directly from the command line), the name of the file to be edited, and where the edited file should be send. The general for of the command line is as follows:

```
sed [-n] [-e commands] [-f scriptname] textfile [>file]
```

The **-n** option will be explained below, in the section on *Output*. The **-e** option tells **sed** that *commands* follow immediately. The **-f** option tells **sed** that the commands are contained in the file *scriptname*. *textfile* is the name of the text file to be edited. The greater-than symbol '>' followed by a file name redirects the edited version of the text file into *file*; if this option is not used, the edited copy of the text file will be sent to the standard output.

In this example, a command script has been prepared, so the **-f** option will be used. Also, the edited text should appear on the terminal screen, so the '>' will not be used. Type the command line as follows:

```
sed -f script1 exercise1
```

The following text will appear on your screen:

```
No man will be a sailor who has contrivance enough
to get himself into a jail; for being in a ship is
being in a jail, with the chance of being drowned.
```

You can use **sed** not only to substitute one word for another, but to add lines to files, delete lines, and perform more involved editing. No matter how complex your **sed** editing becomes, though, **sed** will always use the basic format just described.

The next few sections describe **sed**'s basic commands.

Simple Commands

Type in the exercises exactly as shown and examine the results. Use the **cat** command to enter the command file as well as the input file. The edited text will appear on your terminal. Usually when you edit, you will want to redirect the edited text to a new file; however, for the exercises presented here, let the edited text appear on your terminal so you can examine the results immediately.

Substituting

The substitution command is used very often when editing. **sed**'s substitution command **s** resembles the same command in **ed**. Its form is as follows:

```
s/term1/term2/
```

This tells **sed** to substitute *term2* for *term1*. To correct a misspelled word, for example, use this command form:

```
s/mispel/misspell/
```

As written, this command changes only the first occurrence of **mispel** in each line of your text file. To change *every* occurrence of *mispel* in each line, add **g** (the **g**lobal option) at the end of the command:

```
s/mispel/misspell/g
```

If you want to change only the *third* occurrence of **mispel** on every line, put a **3** after the **s**:

```
s3/mispel/misspell/
```

When no digit follows the **s** and no **g** follows the command, only the first occurrence of the term in each line (should there be one) will be changed.

To practice the substitution, type the following file into your system (please include the misspellings):

```
cat >exercise2
From the Devils Dictionary:
Hemp, n. A plant from whose fiberous bark is made
an article of neckware which is frequently put on
after public speaking in the open air and prevents
the wearer from tking cold.
<ctrl-D>
```

Now, prepare the following **sed** script to correct the misspellings:

```
cat >script2
s/Devils/Devil's/
s/fiberous/fibrous/
s/tking/taking/
<ctrl-D>
```

Invoke **sed** with the following command:

```
sed -f script2 exercise2
```

The following will appear on your screen:

```
From the Devil's Dictionary:
Hemp, n. A plant from whose fiberous bark is made
an article of neckwear which is frequently put on
after public speaking in the open air and prevents
the wearer from taking cold.
```

To see how the **g** command and the number option work, prepare the following text file:

```
cat >exercise3
sd      sd      sd      sd
sd      sd      sd      sd
sd      sd      sd      sd
<ctrl-D>
```

The following **sed** script changes the *third* **sd** in each line to **sed**:

```
cat >script3
s3/sd/sed/
<ctrl-D>
```

Invoke **sed** with the following command line:

```
sed -f script3 exercise3
```

The following will appear on your screen:

```
sd      sd      sed     sd
sd      sd      sed     sd
sd      sd      sed     sd
```

To change *every* **sd** to **sed**, use the **g** option. Prepare the following **sed** script:

```
cat >script3a
s/sd/sed/g
<ctrl-D>
```

The following will appear on your screen:

```
sed     sed     sed     sed
sed     sed     sed     sed
sed     sed     sed     sed
```

The **g** command will be most useful for editing prose, when you have no way to tell how many times a given error will appear on a line. The numeric option will be most useful for editing tables and lists.

Selecting Lines

Each of the substitution commands given above will be applied to every input line. Unlike **ed**, there is no error message if no line of text contains *term1*.

In certain instances, however, you may wish to apply a particular command only to specific lines. Lines can be specified (or *addressed*) by *preceding* the command with the identifying line number. The following exercise demonstrates line selection. First, prepare the following text file:

```
cat >exercise4
When a man is tired of London,
he is tired of life; for there
is in London all that life can afford.
<ctrl-D>
```

To change the word **tired** to **fatigued** on line 2 only, prepare the following **sed** script:

```
cat >script4
2s/tired/fatigued/
<ctrl-D>
```

Begin the editing of your text file by typing the following command line:

```
sed -f script4 exercise4
```

The following will appear on your screen:

```
When a man is tired of London,
he is fatigued of life; for there
is in London all that life can afford.
```

Remember that to specify a line number, you must place the number *before* the command; but to specify the numeric option (that is, position within the line), you must place the number *after* the command.

You can define a *range* of lines to be edited. One way to do this is to list the first and last line numbers, separated by commas, of the block of text in question. For example, the following script will change **is** to **was** only in the first two lines of the text file you just prepared:

```
cat >script4a
1,2s/is/was/
<ctrl-D>
```

Entering the command line

```
sed -f script4a exercise4
```

will bring the following text to your screen:

```
When a man was tired of London,
he was tired of life, for there
is in London all that life can afford.
```

Note that the word **is** in line 3 was unaffected by the substitution command, because it lay outside the range of lines specified by the command.

You can also select lines by *patterns*. Patterns are *strings* (any collection of letters and numbers, such as a word) that can be combined with commands. A fuller description of *patterns* can be found in the tutorial for **ed**. Later on, when we show you other commands, you will see that line selection by pattern rather than by line number is quite useful.

You can use the end-of-file symbol '$' for line selection. When you use this symbol, you do not have to know the exact number of lines in your text file. For example, if you want to apply a substitution command from line 10 through the end of your text file, the command form is:

```
10,$s/term1/term2/
```

p: Print Lines

When **sed** edits a text file, the edited text is by default sent to the *standard output*, which usually is your terminal's screen. (As noted above, the edited text can be optionally redirected to another file by using the shell's '>' operator.) Normally, **sed** prints every line in the text file, whether the line is changed or not.

The next exercise will demonstrate these defaults. First, type in the following text file:

```
cat >exercise5
Bill        g7          r115
Nora        g8          r115
Steve       g7          r120
Ella        g8          r120
Dave        g7          r115
Robert      g8          r120
<ctrl-D>
```

Next, create a script that contains no commands, by typing:

```
cat >script5
<ctrl-D>
```

Now, execute this empty script:

```
sed -f script5 exercise5
```

Note that **sed** simply copied your text file to the screen, without changing it in any way.

This default, however, can be inconvenient if you want to print only a selected portion of a file. Fortunately, with a couple of commands you can control **sed**'s printing.

The command line option **-n** changes **sed**'s printing behavior. When you invoke **-n**, the text file no longer is printed automatically. **sed** prints only the lines specified by the **p** command. The **p** command makes **sed** print whatever line (or lines) to which it is applied. Use **-n** on the command line to stop **sed** from printing every line automatically; then use the **p** command in the script to target the lines you want to print. The following exercise will help you grasp this point. First, type in the following **sed** script:

```
cat >script5a
/g7/p
<ctrl-D>
```

Enter the command line:

```
sed -n -f script5a exercise5
```

and the following text will appear on your terminal:

```
Bill        g7          r115
Steve       g7          r120
Dave        g7          r115
```

sed prints only the records of the students in grade 7 (**g7**).

It is important to note the order, or *syntax*, of the **-n** and **-f** command line options. The correct order is to enter **-n**, then **-f**. (**-nf** or **-fn** are also acceptable.) If you type **-f** and then **-n**, however, all you will get is an error message.

When you use the **p** option with a **sed** command, **sed** will print every line of text in which that command makes a substitution. This can be useful, but if you are not careful it can also create some problems. **sed** normally prints every line in your text file, whether or not it is changed by your script, unless you specify the **-n** option in your command line. Therefore, if you *do not* use the **-n** option in your command line and you **do** use the **p** option with your **s** commands, every line that **sed** edits will be printed more than once.

The following script illustrates this point:

```
cat >script5b
s/g7/g8/gp
s/r115/r120/gp
<ctrl-D>
```

Now, execute it with the following command:

```
sed -f script5b exercise5
```

The result will look like this:

```
Bill        g8          r115
Bill        g8          r120
Bill        g8          r120
Nora        g8          r120
Nora        g8          r120
Steve       g8          r120
Steve       g8          r120
Ella        g8          r120
Dave        g8          r115
Dave        g8          r120
Dave        g8          r120
Robert      g8          r120
```

Bill and **Dave** were printed three times: the first time because the **p** option was specified after editing the grade number, the second time because the **p** option was specified after editing the room number, and the third time because the **-n** option was *not* used on the command line. **Steve** and **Nora** were printed twice: the first time because their lines were edited once each, and the second time because the **-n** option was not used on the command line. **Ella** and **Robert** appeared once because their lines were not edited at all and the **-n** option was not specified in the command line.

To get around this problem, use the **-n** option and use **p** only once, on the last substitution:

```
cat >script5c
s/g7/g8/g
s/r115/r120/gp
<ctrl-D>
```

When you enter the following command line

```
sed -n -f script5c exercise5
```

the new result will be:

```
Bill        g8        r120
Nora        g8        r120
Dave        g8        r120
```

The **w** command acts like the **p** command, except that matched lines are written to the file whose name follows the **w**. The following script shows the correct form:

```
cat >script5d
s/g8/g9/w grade.9
s/gu/g8/w grade.8
<ctrl-D>
```

When you execute script5d with this command:

```
sed -f script5d exercise5
```

the normal product will be seen produced at your terminal, and the edited lines will be written to files **grade.8** and **grade.9**. File **grade.8** will contain:

```
Bill        g8        r115
Steve       g8        r120
Dave        g8        r115
```

Note the order in which the two **s** commands were given. If their order were reversed, every text line with **g7** in it would have **g7** changed to **g8** by the first **s** command, then have this newly created **g8** changed to **g9** by the second **s** command. Thus, *all* the students would be shown to be in **g9**, and every text line would be printed into the file **grade.9**.

Line Location

When you edit a file with **sed**, it is hard to keep track of line numbers. As noted earlier, you can locate specific lines with **sed** by using patterns as *line locators*. To see how this works, type the following text file into your system:

```
cat >exercise6
From the Book of Proverbs:
As a door turneth upon his hinges, so the
slothful man turneth upon his bed.
A soft answer turneth away wrath:  but grievous
words stir up anger.
<ctrl-D>
```

Now, prepare the following **sed** script:

```
cat >script6
/door/,/bed/s/turneth/turns/
<ctrl-D>
```

Execute it by entering the following command line:

```
sed -f script6 exercise6
```

The text will appear on your terminal this way:

```
From the Book of Proverbs:
As a door turns upon his hinges, so the
slothful man turns upon his bed.
A soft answer turneth away wrath:  but grievous
words stir up anger.
```

Note that the word *turns* was substituted for the word *turneth* only in the first proverb, not the second. The reason is that the **s** command in this instance was preceded by the *patterns* **door** and **bed**. These told **sed** to begin making the substitution on the first line in which the word **door** appears, and to stop making the substitution with the first line in which the word **bed** appears. In the text file, the fourth line also contained the word **turneth**, but because it lay outside the range of line specified by the line locators, no substitution was made.

When **sed** locates the last line of a block of text that you have defined, it will immediately look for the next occurrence of the first line locator. If it finds that first line locator, it will then resume making the substitution to your file until it again finds the second line locator or comes to the end of the file, whichever occurs first. In this example, when **sed** found the word **bed**, it began to look again for the word **door**; and if it had found the word **door**, it would have resumed substituting **turns** for **turneth**.

Remember that, as explained earlier, line numbers can also be used as line locators. For example, the **sed** script

```
2,3s/turneth/turns/
```

would have produced the same changes as did the script with the pattern line locators prepared earlier.

Add Lines of Text

sed can add lines to your text file. To see how **sed** does this, first prepare the following text file:

```
cat >exercise7
From the Devil's Dictionary:
Syllogism, n. A logical formula consisting of a major
and a minor assumption and an inconsequent.
<ctrl-D>
```

Now, type in the following script:

```
cat >script7
3a\
Economy, n. Purchasing the barrel of whiskey you do not \
need for the price of the cow you cannot afford.
<ctrl-D>
```

When you implement the script:

```
sed -f script7 exercise7
```

you will see this result:

```
From the Devil's Dictionary:
Syllogism, n. A logical formula consisting of a major
and a minor assumption and an inconsequent.
Economy, n. Purchasing the barrel of whiskey you do not
need for the price of the cow you cannot afford.
```

The append command **a** added text *after* the third line of the file. You defined where the text went. Notice the backslash '\' at the end of the line with the **a** command. This indicates that the next line is part of the command. When you append several lines of text, each line but the last one to be added must end with a '\' as in our example.

Note that no other editing command, such as **s**, can affect any line added with **a**. These lines go directly to your screen, or to a file, should you be sending the edited text there, and are invisible to all other **sed** commands.

The insert command **i** works like the **a** command, except that it adds its lines *before* the addressed line, rather than after. The following script shows how the **i** command works:

```
cat >script7a
2i\
Peace, n. In international affairs, a period of cheating\
between two periods of fighting.
<ctrl-D>
```

Invoking it with this command:

```
sed -f script7a exercise7
```

produces this:

```
From the Devil's Dictionary:
Peace, n. In international affairs, a period of cheating
between two periods of fighting.
Syllogism, n. A logical formula consisting of a major
and a minor assumption and an inconsequent.
```

As with the **a** command, no substitutions or other changes are performed on lines added with **i**.

Note, too, that you can *bracket* a text line by using the **a** and **i** commands at the same time. Adding a line with either **a** or **i** does not change line numbers of the text file you are editing (although it does, of course, change the line numbers of the file **sed** writes).

Delete Lines

The **d** command deletes lines that you do not want in the edited text. The original file stays unchanged, of course.

Lines that match the address (be it a line number, range, or pattern) of a **d** command do not appear in the output. Exercise 8 illustrates the **d** command:

```
cat >exercise8
The sun was shining on the sea,
Shining with all his might.
He did his very best to make
The billows smooth and bright --
And this was odd, because it was
The middle of the night.
<ctrl-D>
```

Now, you have to define the lines to be deleted by matching them with a unique pattern or a line number. To delete lines 3 through 6, prepare this script:

```
cat >script8
/best/,/night/d
<ctrl-D>
```

The command:

```
sed -f script8 exercise8
```

generates this result:

```
The sun was shining on the sea,
Shining with all his might.
```

Note that when a line is deleted, no other commands are applied to it. Usually, if a **sed** script holds a number of commands, every one of those commands is applied to every line read from your text file; however, **sed** is logical enough to read the next text line immediately, should a **d** command delete the current line before the series of commands has finished.

Change Lines

The **c** command combines the **i** and **d** options. Text is inserted before the addressed lines, which are then deleted. To see how this command works, prepare the following text file:

```
cat >exercise9
Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.
<ctrl-D>
```

Now, type in the following script:

```
cat >script9
1,2c\
Twas brilliant, and the shining cove\
Did glare and glimmer in the wave;
<ctrl-D>
```

When you execute your script with the following command line:

```
sed -f script9 exercise9
```

the result is:

```
Twas brilliant, and the shining cove
Did glare and glimmer in the wave;
All mimsy were the borogoves,
And the mome raths outgrabe.
```

Like the **i** and **a** commands, the **c** command requires all added lines but the last to end with '\'.

Include Lines From a File

When you edit a file, you may wish to include, or *read in*, a second file as part of it. This is done with **r** command. To see how this works, type the following file into your computer, and call it **include**:

```
cat >include
      Then there comes the often-used refrain
      Whose repetitious writing dulls the brain.
<ctrl-D>
```

Now, prepare the file to be edited:

```
cat >exercise10
To write a poem doesn't take much time;
Just string some words to rhythm and a rhyme.
What poets do to language is a crime,
Words and syntax twisted for a rhyme.
<ctrl-D>
```

When you write your script, you must tell **sed** where to read in **include**. The form of the command should be familiar by now:

```
cat >script10
/rhyme/r include
<ctrl-D>
```

The result is:

```
To write a poem doesn't take much time;
Just string some words to rhythm and a rhyme.
      Then there comes the often-used refrain
      Whose repetitious writing dulls the brain.
What poets do to language is a crime,
Words and syntax twisting for a rhyme.
      Then there comes the often-used refrain
      Whose repetitious writing dulls the brain.
```

Note that the **r** command inserted **include** *after* the addressed line. You can address lines by number, of course, as well as by pattern.

Quit Processing

The **q** command makes **sed** stop processing the text file. You will use this command most often to limit the application your **sed** script to a portion of your text file. For example, if you were editing a large file and you knew that your commands would be irrelevant to the last half of the file, you could insert an appropriately addressed **q** and save some computer time. You can also use this command to print portions of a file.

To see how this is done, prepare the following text file:

```
cat >exercise11
An hourglass has a very wide top,
a very narrow
middle
and a bottom
that is also extremely wide.
<ctrl-D>
```

The following script will print the top of the text file. Note how the script uses **middle** to address the line where the file is to be split.

```
cat >script11
/middle/q
<ctrl-D>
```

The command:

```
sed -f script11 exercise11
```

produces:

```
An hourglass has a very wide top,
a very narrow
middle
```

To print out only the lines *after* the pattern **middle**, simply delete the first half of the file with the **d** command, as follows:

```
cat >script11a
1,/middle/d
<ctrl-D>
```

The result is the output:

```
and a bottom
that is also extremely wide.
```

Next Line

The **n** command advances to the next line of the text file. The **n** command is useful for instances when you have two or more interrelated lines, and you want to ensure th a particular set of patterns is matched over the entire set of lines. To see how **n** works, prepare the following text file:

```
cat >exercise12
Alpha
One
Beta
Two
Gamma
Three
Delta
Four
Epsilon
Five
<ctrl-D>
```

To print a list of letters alone, type the following script:

```
cat >script12
n
d
<ctrl-D>
```

and execute it with the following command line:

```
sed -f script12 exercise12
```

The result will be the following:

```
Alpha
Beta
Gamma
Delta
Epsilon
```

Remember that **n** does *not* stop processing, go to the next text line, and begin processing all over again. Rather, it simply reads the next input line and continues processing from where it left off. For example, if your **sed** file consisted of three commands, the second of which was the **n** command, **sed** would apply the first command to the first line it read, then jump to the second line and apply the last commands. Then, it would read the third line and begin the pattern over again. To see how this works, prepare the following text file:

```
cat >exercise13
Alpha
Alpha
Alpha
Alpha
<ctrl-D>
```

Now type in this script:

```
cat >script13
s/Alpha/Apple/
/Apple/n
s/Alpha/Banana/
<ctrl-D>
```

When you execute the script with this command line:

```
sed -f script13 script13
```

the following will appear on your terminal:

```
Apple
Banana
Apple
Banana
```

Note that the first substitution command changed the first **Alpha** to **Apple**; the **n** command moved **sed** to the next line; and the second **s** command changed that **Alpha** to **Banana**.

Advanced sed Commands

The following sections discuss **sed**'s advanced features. They also discuss the method of operation.

Work Area

As described earlier, **sed** reads your text file one line at a time, and applies all of your editing commands to that line. After the editing commands have been applied, the edited line is either sent to the *standard output*, written to a file you have named, or thrown away, depending on what you have told **sed** to do.

When **sed** reads a line from your text file, it copies that line into a *work area*, where it actually executes your editing commands. **sed** notes the number of the line in the work area, then executes each editing command in turn, first checking to see if the patterns or line numbers specified in each command actually apply to that line. After each command is checked in turn and performed if indicated, **sed** prints the edited line (if it is supposed to be), and reads the next text line.

Add to Work Area

The exercises so far have used only one line in the work area. The **N** command, however, tells **sed** to read a second line into the work area. The following exercise illustrates the use of the work area and the **N** command.

```
cat >exercise14
This exercise has a brok
en word.
<ctrl-D>
```

Now, prepare the following **sed** script:

```
cat >script14
/brok$/N
s/brok\nen/broken/
s/has/had/
<ctrl-D>
```

and execute it with the following command line:

```
sed -f script14 exercise14
```

which produces the following text:

```
This exercise had a broken sentence.
```

You will find it helpful to review this exercise in some detail. The first command in the script

```
/brok$/N
```

tells **sed** to search for the pattern **brok** at the *end* of the line of text. (The dollar sign '$' in this instance indicates the end of the line; remember that when the '$' is used with a line number, it indicates the end of the *file*.) The **N** command tells **sed** to keep this line in the working space, and copy the *next* line into the working space as well.

When **sed** executes this command on the present text file, the work area will look like this:

```
This example has a brok<newline>en word.
```

Note that the two lines now appear to **sed** as though they formed one long line. The word **<newline>** represents the end of line character that tells your terminal or printer to jump to a new line when the text file is printed out. This character is invisible, but it is there, and it can be changed or deleted. You can describe this character to **sed** by using the characters **\n**. The first substitution in this script

```
s/brok\nen/broken/
```

replaces **brok<newline>en** with **broken**. Because the newline character is deleted from the text, what used to be printed out as two lines on your screen will now be printed out as one.

Note the difference, too, between the **n** and **N** commands. The **n** command will *replace* the text line in the work area with the next line from your text file. The **N** command, however, *appends* the next line from your text file to the end of the text already in the working area. The next exercise demonstrates this difference. First, create the following text file:

```
cat >exercise15
Apple
Apple
Apple
Apple
<ctrl-D>
```

Now, prepare the following two scripts:

```
cat >script15
/Apple/n
s/Apple/Banana/g
<ctrl-D>

cat >script15a
/Apple/N
s/Apple/Banana/g
<ctrl-D>
```

When script15 is executed with the following command line:

```
sed -f script15 exercise15
```

this will appear on your screen:

```
Apple
Banana
Apple
Banana
```

The **n** command told **sed** to print out the line already in the work area before reading in the next line from the text file. This meant that **sed** substituted **Banana** for **Apple** only on the *second* line of each pair.

Note what happens, however, when you run script15a, using this command line:

```
sed -f script15a exercise15
```

This text appears:

```
Banana
Banana
Banana
Banana
```

Because *both* lines of each pair were kept in the work area, the substitution command changed both of them.

Print First Line

The **P** command prints material from the work area. Unlike the **p** command, which prints *everything* in the work area, **P** prints only the *first* line in the work area. To see how this works, prepare the following text file:

```
cat >exercise16
Student:  George
Teacher:  Mr. Starzynski
Student:  Marian
Teacher:  Miss Peterson
Student:  Ivan
Teacher:  Mr. Starzynski
<ctrl-D>
```

Now, prepare the following scripts:

```
cat >script16
/Student/N
/Mr. Starzynski/p
<ctrl-D>

cat >script16a
/Student/N
/Miss Peterson/P
<ctrl-D>
```

When the first of these scripts is executed with the following command line (note the use of the **-n** option):

```
sed -n -f script16 exercise16
```

the result is

```
Student:  George
Teacher:  Mr. Starzynski
Student:  Ivan
Teacher:  Mr. Starzynski
```

whereas script16a, when executed as follows:

```
sed -n -f script16a exercise16
```

produces

```
Student:  George
Student:  Ivan
```

Note that the **N** command lines pull both the name of the student and the name of the teacher into **sed**'s work area; then the **P** command allows you to print only the names of the students whose teacher is Mr. Starzynski. Obviously, **P** is a powerful tool that will allow you to select material from tables, lists, and other repetitive files.

Save Work Area

sed can create a second work area in addition to the primary work area in which **sed** performs its editing. **sed** does not execute any editing commands on the material stored in this secondary work area; rather, this work area can be used to store material that you want to edit or insert later.

The commands **h** and **H** copy material from the primary work area into the secondary work area. **h** and **H** differ in that **h** *displaces* any material in the secondary work area with the line being copied there, whereas **H** *appends* the line being copied onto the material already in the work area. Note, too, that both **h** and **H** merely *copy* the primary work area into the secondary work area — after these commands have been executed, the material in the primary work area remains intact, and can be edited further, printed out, or deleted, whichever you prefer.

The commands **g** and **G** copy material back from the secondary work area into the primary work area. Again, these commands differ in that **g** *displaces* whatever is in the primary work area with the material from the secondary work area, whereas **G** *appends* the material from the secondary work area onto the material already in the primary work area.

The following exercises will demonstrate how these commands are used. First, create the following text file:

```
cat >exercise17
fruit:  apple
berry:  gooseberry
fruit:  orange
berry:  raspberry
fruit:  pear
berry:  blueberry
<ctrl-D>
```

The first script uses the **h** and **g** commands:

```
cat >script17
/fruit/h
/fruit/d
/berry/g
<ctrl-D>
```

When you execute this script with the following command line:

```
sed -f script17 exercise17
```

you receive the following text on your screen:

```
fruit:  apple
fruit:  orange
fruit:  pear
```

Review the last script in detail. The first command, **/fruit/h**, copied the line beginning with "fruit" into the secondary work area, displacing whatever happened to be there. The command **/fruit/d** then deleted the line from the primary work area; if this were not done, it would then have been printed out. The third command, **/berry/g** then recopied the material from the secondary work area into the primary work area, displacing whatever was already in the primary work area. The result of all this shuffling and displacing was that the three lines that begin with **fruit** were printed out.

The next script demonstrates the **H** command:

```
cat >script17a
/fruit/H
/fruit/d
/berry/g
<ctrl-D>
```

When you execute this script with the following command line:

```
sed -f script17a exercise 17
```

you see:

```
fruit:  apple
fruit:  apple
fruit:  orange
fruit:  apple
fruit:  orange
fruit:  pear
```

Because the **H** command *appends* material into the secondary work area, rather than replacing it as **h** does, all three lines that began with **fruit** were cumulatively stored in the secondary work area. Because the **g** command was used for every line that began with **berry**, the contents of the secondary work area (that is, the **fruit** lines) were written over each of the three lines that began with **berry**.

The next script demonstrates the use of the **G** command:

```
cat >script17b
/fruit/H
/fruit/d
/berry/G
s/berry:/g
s/fruit:/g
<ctrl-D>
```

When you execute this script with the following command line:

```
sed -f script17b exercise17
```

you will see:

```
gooseberry
  apple
raspberry
  apple
  orange
blueberry
  apple
  orange
  pear
```

The **H** command copies the lines that begin with **fruit** into the secondary work area. The **G** command then re-copies them from the secondary work area into the primary work area, and appends them to the material already in the primary work area — that is, to a line that begins with **berry**.

The two substitution commands then strip off the **fruit** and **berry** prefixes; obviously, these substitutions do not affect the operation of the **H** and **G** commands, but they do create a tidier result.

By the way, be sure you distinguish the **g** command from the **g** option used with the **s** command. If you do not, what **sed** finally prints out for you may appear very strange.

The final command that uses the secondary work area is **x**, which exchanges the two work areas. The following script shows how this is used:

```
cat >script17c
/fruit/H
/fruit/d
/blueberry/x
s/berry:/g
s/fruit:/g
<ctrl-D>
```

When you execute this script with the following command line:

```
sed -f script17c exercise17
```

you see:

```
gooseberry
raspberry
  apple
  orange
  pear
```

The text lines that began with **fruit** were moved into the secondary working area. The **x** command was executed when the line that contained the word **blueberry** was reached, and the two working areas exchanged their contents. The **fruit** lines were then printed out, while the **blueberry** line was simply left in the secondary working at the end of the program, and disappeared when the program concluded.

Note that **x** simply swaps the two working areas — there is no "**X**" command that appends the work areas onto each other.

Transform Characters

The **y** command is a special form of the **s** command. With the **y** command, you can replace a number of characters easily, without having to write a series of **s** commands.

The form of the command is:

```
y/123/abc/
```

In the above example, **1** will be replaced with **a**, **2** with **b**, and **3** with **c** throughout the document (no **g** option is needed). For **y** to work properly there must be a one-to-one relationship between the characters being replaced and the characters replacing them. Also, **y** cannot make exchanges that involve more than one character — it cannot, for example, replace **apple** with **banana**.

One useful task for the **y** command is to change all upper-case letters in a file to lower case. Prepare the following text file to see how this is done:

```
cat >exercise18
NOW IS THE TIME FOR ALL GOOD MEN TO COME
TO THE AID OF THE PARTY.
<ctrl-D>
```

And prepare the following script, which will change these capitals:

```
cat >script18
y/ABCDEFGHI/abcdefghi/
y/JKLMNOPQR/jlkmnopqr/
y/STUVWXYZ/stuvwxyz/
<ctrl-D>
```

The alphabet is entered here in three chunks, to prevent the command from being too long to type easily. Execute this script with the following command line:

```
sed -f script18 exercise18
```

The result is:

```
now is the time for all good men to come
to the aid of the party.
```

Command Control

sed gives you advanced control over the execution of commands. The next subsections describe how these command controls help you write compact, powerful scripts.

{ }: Command Grouping

In several of the exercises presented so far, more than one command specified the same line locator. By using braces '{' and '}', you can bundle commands, which makes writing your scripts easier and lessens the chance of making a typographical error.

To see how this is done, type the following script:

```
cat >exercise19
When my love swears that she is made of truth,
I do believe her, though I know she lies,
That she might think me some untutored youth,
Unlearned in the world's false subtleties.
<ctrl-D>
```

Now, prepare the following script:

```
cat >script19
/truth/{N
P
}
/lies/d
<ctrl-D>
```

When you execute this script with the following command line:

```
sed -f script19 exercise19
```

the result on your terminal is:

```
When my love swears that she is made of truth,
That she might think me some untutored youth,
Unlearned in the world's false subtleties.
```

Note the syntax of this command. Each subsequent command must go on a line of its own, as must the right brace '}'. If this syntax is not observed, you will receive an error message.

!: All But

The **!** flag inverts a line selector; that is to say, the command will be performed on every line *but* the one named in the line selector. The following script will show how this works:

```
cat >script19a
2!d
<ctrl-D>
```

which, when run with the following command line:

```
sed -f script19a exercise19
```

produces

```
I do believe her, though I know she lies,
```

This script deleted every line *except* line 2. The **!** flag may also be used with a range of lines, as indicated by line numbers or line patterns; in either case, you must place the **!** flag *after* the line selectors and immediately *before* the command. Obviously, the **!** flag is very powerful, and can be used to sift out a few desired lines from a large file.

= : Print Line Number

You may wish to print only the *line number* of lines that contain a selected pattern. This is done with the **=** command. For example, you may wish to know the number of each line in the exercise that contains the word **she**. The following script:

```
cat >script19b
/she/=
<ctrl-D>
```

when executed with the following command line (note the **-n** option):

```
sed -n -f script19b exercise19
```

produces this result:

```
1
2
3
```

These numbers can be stored in a file and used in further editing, or included with the text of the fully edited file to provide a series of line markers.

Skipping Commands

sed normally processes editing commands in order, beginning with the first command and proceeding sequentially to the last. This behavior can be modified by the branching commands: **b**, **t**, and **:**.

These commands must be used with the colon (**:**) command, which defines a *label* point in the list of commands.

The **branch** command **b** allows you to skip unconditionally some editing commands in your script. The following exercise demonstrates how this can be used:

```
cat >exercise20
They went to sea in a sieve, they did;
In a sieve they went to sea;
In spite of all their friends could say,
On a winter's morn, on a stormy day,
In a sieve they went to sea.
<ctrl-D>
```

The following script uses the **b** command to avoid making certain changes to the first line of the poem:

```
cat >script20
s/sea/drink/g
/They/bend
s/sieve/ship/g
:end
```

When you execute this script with the following command line:

```
sed -f script20 exercise20
```

you will see:

```
They went to drink in a sieve, they did;
In a ship they went to drink;
In spite of all their friends could say,
On a winter's morn, on a stormy day,
In a ship they went to drink.
```

Note that the word **sea** is changed to **drink** throughout the file; however, when **sed** noted that the word **They** appeared in line 1, the **b** command forced it to seek the **:** command that was labeled with the word **end**, and to continue editing only *after* it found the labelled **:** command. In so doing, **sed** skipped the command to substitute **ship** for **sieve**, which is why that substitution was not made in line 1.

Note the syntax of the **b** command: the label follows it without a break. The text of the label is unimportant, just so long as it matches that used in the **b** command; however, the use of a label allows you to place several **b** or (as will be seen) **t** commands in the same script without mixing them up.

t: Test Command

The **test** command, **t**, also allows you to change the order in which editing commands are executed. Unlike the **b** command, which simply examines a line for a given pattern, the **t** command *tests* to see if a particular substitution has been performed.

The following script demonstrates the use of the **t** command:

```
cat >script20a
s/They/they/g
tend
s/sieve/ship/
:end
s/sea/drink/g
<ctrl-D>
```

which, when executed with the following command line:

```
sed -f script20a exercise20
```

produces:

```
they went to drink in a sieve, they did;
In a ship they went to drink;
In spite of all their friends could say,
On a winter's morn, on a stormy day,
In a ship they went to drink.
```

Note that the **t** command checked to see that **they** was substituted for **They** before branching to the ':' command labeled with the word **end**.

Also note the syntax of the **t** command: Like the **b** command, the label immediately follows the command and is not separated by a space; unlike the **b** command, however, the **t** command appears on the line *below* the substitution command for which it is testing.

For More Information

The Lexicon entry for **sed** summarizes its command-line options and commands. The COHERENT line editor **ed** resembles **sed**, except that it works interactively instead of in a stream. For information on **ed**, see its tutorial or its entry in the Lexicon.

Section 13:

Introducing sh, the Bourne Shell

sh is the command that invokes the Bourne shell, which is the COHERENT system's default command interpreter. The Bourne shell interpets commands, expands file names in various sophisticated ways, permits conditional execution of commands, and much more. The Bourne shell is, in effect, both a programming language and an interpreter.

At least one writer has noted that the shell is the original "fourth-generation language" — that is, a powerful programming language that is straightforward enough to be programmed by non-programmers. You will find that taking a little time to master the rudiments of the shell programming language will pay enormous benefits in making best use of your COHERENT system.

Simple Commands

The shell command language is built around simple commands. For example, the following command lists all files in the current directory:

```
lc
```

You can combine several simple commands on one line by separating them with semicolons:

```
who;du;mail
```

The shell executes the commands in sequence as if they had been typed:

```
who
du
mail
```

In both of these examples, **du** does not begin execution until **who** is finished, and **mail** does not begin until **du** is done.

Special Characters

The shell treats the following characters specially; if you want to use them without their special meaning, you must precede them with the backslash character \, or enclose them within quotation marks:

```
* ? [ ] | ; { } ( )
$ = : ' ' " < > << >>
```

The function of these characters will be explained later in this section. To use one of these characters in a command, for example '?', type:

```
echo \?
```

In addition, the shell treats the following words in a special way when they appear as the first word of a command:

```
case do done elif else esac
fi if in then until while
```

Running Commands in the Background

The shell can execute commands simultaneously as well as sequentially. This means that while the shell is executing one command, it lets you type and execute another command. Under the shell, the number of commands you can execute at the same time is limited mainly by the amount of memory and disk space on your system.

If a command is followed by the special character '&', the shell begins to execute it immediately, and prompts you to enter another command. For example, if you need to **sort** a large file but want to continue with other commands while the sort is executing, you can type:

```
sort >bigfile.sorted bigfile.unsorted &
ed prog
```

This allows you to edit file **prog** while your computer quietly executes the sort in the background.

When you run a command with **&**, the shell types the *process id* of the command started in background. When the COHERENT system runs a command, it assigns that command a *process id*, which is a number that uniquely identifies that command to COHERENT. Normally, there is no need to be concerned about these numbers. However, when you run commands in the background, the shell tells you the id of the background process so you can keep track of its execution.

The command

```
ps
```

lists the processes you are currently running. If you have no background jobs, the response is:

```
TTY PID
30: 362 -sh
30: 399 ps
```

The first column shows the number that COHERENT has assigned to your terminal. This is the same terminal number printed out by **who**. The second column shows the process id; the third column shows the program or command executing. The characters **-sh** in the third column means the login shell. There are two processes because the shell is running the **ps** command as a separate process.

Once you have started a background command, **ps** shows you the process entry, which has the process id that the shell typed out for you.

If you need the results from a background job, you can wait for it to finish by issuing the command:

```
wait
```

The shell will then accept no further commands until all your background jobs are finished. If there are no background jobs, there will be no delay.

Scripts

Many of the commands that you use in COHERENT are *programs*, such as **ed**. Others, like the **man** command, are *scripts*, or files that merely contain calls to other commands. You can write scripts on your own, simply by using a text editor to type into a file the commands you wish to execute. If you frequently use a set of commands, you can save yourself from having to type them over and over by simply typing them once into a script.

For example, suppose that you wish to check periodically the amount of disk space that you have used, the amount of disk space still available, and see who is using the system. You can write a script to do all of this automatically. Create the script **good.am** by typing the following commands:

```
ed
a
du
df
who | sort
mail
.
w good.am
q
```

From now on, to execute the above-listed commands, you need only type:

```
sh good.am
```

where **sh** is a command that means: read commands from a file, in this case **good.am**. If you can issue a command from your terminal, you can also execute it from within a script.

You can make a command file directly executable by using the command **chmod**. For example, the command

```
chmod +x good.am
```

lets you execute the script **good.am** by typing

```
good.am
```

and leaving off the **sh**. Once you have done the **chmod** command, you can still issue the commands by typing:

```
sh good.am
```

as well as use **ed** or MicroEMACS to change the contents of the script.

Notice that the commands called by a script may themselves be scripts. This is illustrated by the following script, **second.sh**:

```
ed
a
sh good.am
lc
.
w second.sh
q
```

Thus, typing:

```
sh second.sh
```

calls the script **good.am**,and also calls the command **lc**.

.profile: Login Shell Script

When you log into the system and before you are issued your first prompt, COHERENT checks your home directory for a file named **.profile**;if it is present, the shell executes the commands it contains.

This enables you to have COHERENT execute commands as soon as you log in. Check if your installation provides one for you by doing an **lc**(be sure that your current directory is the home directory). If the file is there, print it by saying:

```
cat .profile
```

Some of the commands may be of the form:

```
PATH=':/bin:/usr/bin'
```

This sort of command will be discussed below.

Substitutions

Scripts of the form shown above are processed by the COHERENT shell without change. However, the COHERENT shell increases the power of commands by performing three kinds of substitutions within commands before it executes them.

First, it replaces special characters in commands with file names from the current or other directories. This allows you to issue a single command that processes several files.

Second, you can give a script *arguments*, much like arguments that are passed to a Pascal, Algol, or C procedure. This lets you target the action of the script to a specific file name specified when you call it.

Third, the output of one command can be "piped" into another command to serve as its input.

We will use the command **echo**to illustrate these kinds of substitution. Remember that substitutions take place for all commands in the same way that they do for **echo**.

File Name Substitution

File names are often used as command parameters. That is, you will often tell a command to do something to one or more files. By using special shell characters, you can substitute file names in commands. These special characters describe file name *patterns* for the shell to look for in the directory. When the shell finds the file names, it replaces the pattern with them.

The asterisk *****matches any number of any characters in file names. Thus,

```
echo *
```

echoes all the file names in the current directory, whereas

```
echo f*
```

gives all file names that begin with the letter **f**,and

```
echo a*z
```

lists all names that begin with **a** and end with **z**.

To illustrate more clearly, create two files by typing

```
cat >zz1
<ctrl-D>
cat >zz2
<ctrl-D>
```

Then the **echo** command

```
echo zz*
```

produces the output:

```
zz1 zz2
```

Thus, by using a single *****, you can substitute several file names into a command. In other words, the command

```
echo zz*
```

is equivalent to

```
echo zz1 zz2
```

If no file names fit the pattern, the special characters are not changed, but left in the command exactly as you typed them. To illustrate, type the command

```
rm zz*
echo zz*
```

The first command will remove all files whose names begin with **zz**, and is therefore equivalent to:

```
rm zz1 zz2
```

The **echo** command that follows, however, echoes

```
zz*
```

because no files begin with **zz**; they were just removed.

Enclosing command words within apostrophes prevents the shell from matching file names with the enclosed characters. In the unlikely event that you have a file whose name is

```
zz*
```

that you want to remove, use the command

```
rm 'zz*'
```

The ***** is enclosed within apostrophes, and therefore is not changed by the shell.

Another special character **?** match any one letter. To see how this works, create empty files **file1**, **file2**, and **file33** by typing:

```
>file1
>file2
>file33
```

The command

```
echo file?
```

replies

```
file1 file2
```

because **?** does not match **33**.

You can use brackets **[** and **]** to indicate a choice of single characters in a pattern:

```
echo file[12]
```

This command replies:

```
file1 file2
```

To match a range of characters, separate the beginning and end of the range with a hyphen. The command

```
echo [a-m]*
```

prints any file name beginning with a lower-case letter from the first half of the alphabet, and is exactly equivalent to:

```
echo [abcdefghijklm]*
```

When such patterns find several file names, they are inserted in alphabetical order.

Because the character **/** is important in path names, the shell does not match it with ***** or **?** in patterns. The slash must be matched explicitly; that is, it is matched only by a **/** itself. Therefore, to find all the files in the **/usr** directories with the name **notes**, type:

```
echo /usr/*/notes
```

The asterisk matches all the subdirectories of **/usr** that contain a file named **notes**.

In addition, a leading period in a file name must be matched explicitly. If you have a file in your current directory with the name **.profile**, the command

```
echo *file
```

does not match it.

These patterns can appear anywhere within a command or a command file.

Parameter Substitution

Each shell script can have up to nine *positional parameters*. This lets you write scripts that can be used for many circumstances. Recall that command parameters follow the command itself and are separated by tabs or spaces. An example of a command reference with two parameters is:

```
show first second
```

where **first** and **second** are the parameters.

To substitute the positional parameters in the script, use the character **$** followed by the decimal number of the parameter. For example, build the script **show** by typing:

```
ed
a
cat $1
cat $2
diff $1 $2
.
w show
q
chmod +x show
```

$1 and **$2** refer to the first and second parameters, respectively. Create two sample files:

```
cat >first
line 1
line two
line 3
<ctrl-D>
cat >second
line 1
line 2
line 3
<ctrl-D>
```

Then, issue the **show** command

```
show first second
```

which has the same effect as typing:

```
cat first
cat second
diff first second
```

If you issue the **show** command with fewer than the required number of parameters, the shell substitutes an empty string in its place. For example, using the command with only one parameter

```
show first
```

is equivalent to

```
cat first
cat
diff first
```

where the null string has been substituted for **$2**.

The example above shows the parameter references separated from each other by a space. In some uses, you may wish to prefix a substituted parameter to a name or a number. When more than one digit follows a **$**, the shell picks up the first digit as the number of the parameter. To illustrate, build a shell file **pos**:

```
ed
a
echo $167
.
w pos
q
chmod +x pos
```

Then call the script with

```
pos five
```

and the result will be:

```
five67
```

Shell Variable Substitution

In addition to positional parameters, the shell provides *variables*. You can assign values to variables, test them, and substitute them in commands.

The variable name can be built from letters, numbers, and the underscore character; for example:

```
high_tension
a
directory
167
```

Note that keywords must not be single digits, because the shell then treats them as positional parameters. Be aware that the shell treats upper-case and lower-case letters differently in variable names.

An assignment statement gives a value to a shell variable:

```
a=welcome
```

You can inspect their value with the **echo** command:

```
echo $a
```

The shell substitutes the value of the variable **a** in the **echo** command, which then appears as

```
echo welcome
```

COHERENT responds to this command by printing:

```
welcome
```

Don't forget the **$** when referring to the value.

Notice that the shell looks for special characters in any command that it sees — this includes the *space* character. To avoid problems, enclose the value to be assigned in apostrophes:

```
phrase='several words long'
```

There are several uses for variables. One is to hold a long string that you expect to type

repeatedly as part of a command. If you are editing files in a subdirectory like

```
/usr/wisdom/source/widget
```

you can abbreviate if you set a variable **pw** to:

```
pw='/usr/wisdom/source/widget'
```

Then simply using **$pw** in a command

```
echo $pw
```

substitutes the long path name.

Another use of shell variables is as keyword parameters to commands. These then can be used the same way as positional parameters. To see how this works, create another script resembling **show**:

```
ed
a
cat $one
cat $two
diff $one $two
.
w show2
q
chmod +x show2
```

To use **show2**, issue:

```
one=first two=second show2
```

This is equivalent in effect to:

```
cat first
cat second
diff first second
```

Unlike positional parameters, keyword parameters may be several characters in length. If you want some text to follow immediately a keyword parameter, enclose the keyword parameter in braces. To illustrate this, build a command file called **brace**, as follows:

```
ed
a
echo 'with brace:' ${a}bc
echo 'without brace:' $abc
.
w brace
q
chmod +x brace
```

Call the command file with **a** set:

```
a=567 brace
```

The result is:

```
with brace: 567bc
without brace:
```

When used in this way, the keyword parameters must be assigned before the command and on the same line as the command. In this case, the assignment of keyword parameters does not affect the variable after the command is executed. For example, if you type:

```
one=ordinal
one=first two=second show2
echo 'value of one is ' $one
```

echo produces:

```
value of one is ordinal
```

Variables set other than on the line of a command are not normally accessible to a script. To illustrate, build a parameter display script:

```
ed
a
echo 1 $1 2 $2 p1 $p1 p2 $p2
.
w pars
q
chmod +x pars
```

This will be used to show the behavior of parameters. The parameters to **echo**without a **$** help to read the output. To pass positional parameters, type:

```
pars ay bee
```

The output is:

```
1 ay 2 bee p1 p2
```

To pass keyword parameters, type:

```
p1=start p2=begin pars
```

The result is:

```
1 2 p1 start p2 begin
```

To illustrate that the setting of **p1**and **p2**did not "stick", type:

```
echo $p1 $p2 'to show'
```

echoreplies:

```
to show
```

This indicates that **p1**and **p2**are not set.

Illustrating that variables set separately from a command are not seen by the command, type:

```
p1=outside1 p2=outside2
pars
```

This replies:

```
1 2 p1 p2
```

By using the **export** command, however, such variables can be made available to commands. The commands

```
export p1 p2
p1='see me' p2=hello
pars
```

produce:

```
1 2 p1 see me p2 hello
```

This indicates that after the **export** of **p1** and **p2**, they are available to other commands. Once a variable has appeared in an **export** command, its value can be changed without a need to **export** it again.

Command Substitution

By enclosing a command between ` characters, you can feed its output onto the command line of another command. For example

```
echo `ls`
```

echoes the output of the **ls** command.

Special Shell Variables

When you log into the COHERENT system, it sets the shell variable **HOME** to your *home* or default directory path. If your user name is **henry**, then the command

```
echo $HOME
```

on most systems prints:

```
/usr/henry
```

The change directory command **cd** sets the working directory to the path found in **HOME** if no argument is given.

The shell normally prompts you with **$** for commands, and with **>** if more information is needed. These two prompts are taken by the shell from the variables **PS1** and **PS2**. You can change these if you want different prompts, for example

```
PS1="Fred's Software Palace: "
PS2='!'
```

To have these take effect each time you log in, put the assignment statements in your **.profile** file.

The shell variable **PATH** lists the path names of directories that contain commands. To show the contents of **PATH**, type:

```
echo $PATH
```

It typically will show:

```
:/bin:/usr/bin
```

This means that the shell looks for a command first in the current directory, then in **/bin**,

and, if not found there, then in **/usr/bin**. The path names are separated by ':'. This means that an empty string precedes the first ':', the current directory. Another common setting for **PATH** is:

```
:..:/bin:/usr/bin
```

This means that the shell seeks commands first in the current directory, then in '..' (the parent directory of the current directory), then in **/bin**, and finally in **/usr/bin**.

dot . : Read Commands

Similar to the command **sh** is the **.** command. The command

```
. cfil
```

causes the shell to read and execute commands from **cfil**.

This differs from the **sh** command in several respects. First, there's no way to pass parameters to **cfil** with the '.' command. Second, the **sh** command executes another shell to read the commands, whereas '.' simply reads the commands directly. Finally, all the string variables and parameters are accessible by **cfil**.

The command file **good.am** created earlier can be executed with:

```
. good.am
```

This has the same effect. Similarly, the '.' can be itself be used within a command file:

```
ed
a
. good.am
lc
.
w third.sh
q
```

Then, the command

```
. third.sh
```

has the same result as the command:

```
sh third.sh
```

Values Returned by Commands

Most COHERENT commands return a value that indicates success or failure. For example, if **grep** cannot find your file, it issues a diagnostic message and returns a value that tells the shell that something went wrong. You can examine this value by typing the command:

```
echo $?
```

This tells you the value returned by the last command executed. Zero indicates success (true), whereas a non-zero value indicates failure (false). Note that this convention is the opposite of that in the C language (a fact that has led to confusion on occasion).

You can use the value returned by a command to affect decisions about executing other commands.

test: Condition Testing

For most commands, the return value is a side-effect of their operation. However, the **test** command's only task is to return a value. This command can test many conditions, and return a value to indicate whether the requested condition is true or false.

To determine if a file exists, the command

```
test -f file01
```

returns true (zero) if **file01** exists and is not a directory. To check if a file is a directory, use:

```
test -d file01
```

test can also test strings. This is useful when you are using parameter substitution. To illustrate, build the following command:

```
ed
a
test $1 = $2
echo 'test 1 & 2 for equal:' $?
test $1 != $2
echo 'test 1 & 2 for not equal:' $?
.
w test.ed
q
chmod +x test.ed
```

Because the '=' is a parameter, be sure to surround it with space characters.

This command file tests its two parameters for equality. Try the commands:

```
test.ed one two
test.ed one one
```

The **test** command has many other options; see the Lexicon entry for **test** for details.

Executing Commands Conditionally

Type the following commands to create two files:

```
cat >file1
line one
line two
line three
<ctrl-D>
cat >file2
line one
two is different
line three
<ctrl-D>
```

Now, compare the files and print the return value:

```
cmp -s file1 file2
echo $?
```

The command **cmp** compares two files byte-by-byte; the **-s** option tells **cmp** merely to indicate whether the files were the same. This prints **1** (false) because the files are not the same.

To process a second command based on the result returned by the first, type:

```
cmp -s file1 file2 || cat file2
```

The characters **||** signify that the following command **cat** should be executed if the **cmp** command returns a non-zero value, which it will for this example.

The two characters **&&** execute the command that follows them only if the preceding command returns true (zero).

To see how this works, create a third file with the command:

```
cp file1 file3
```

Type the command:

```
cmp -s file1 file3 && rm file3
```

This command removes **file3** if **cmp** indicates that **file1** and **file3** are identical. Because **cmp** is preceded by the copy command **cp**, the files **file1** and **file3** are identical, and so **file3** is removed.

Control Flow

Because the shell is a programming language as well as a program, it provides constructs for conditional execution and loops. These are **for**, **if**, **while**, **until**, and **case**. Also, a subshell can be executed within '(' and ')'.

for: Execute a Loop

The **for** construct processes a set of commands once for each element in a list of items.

To illustrate **for**, type the following commands to COHERENT:

```
for i in a b c
do echo $i
done
```

The items **a**, **b**, and **c** form the list of value that the variable **i** assumes. The shell executes **echo** with **i** assuming each value in turn. The result of these commands is:

```
a
b
c
```

Notice that after you type the line containing **for**, COHERENT prompts with a different character **>** (on most COHERENT systems). The shell does this to remind you that you must type more information. After you type the line containing **done**, the prompt again becomes **$**.

The **for** command is usually used within a script. Also, you can leave off the list of value to the index variable; when you do this, the shell by default uses the arguments typed on the script's command line as the values for the index variable. To illustrate, type:

```
ed
a
for i
do echo $i
echo '---'
done
.
w script.for
q
chmod +x script.for
```

The

```
for i
```

statement is equivalent to:

```
for i in $*
```

where **$*** means "all positional parameters". Notice that two commands are repeated for each value of **i**. Now, call **script.for** with the following command line:

```
script.for 1 2 3 4 test
```

The result is:

```
1
---
2
---
3
---
4
---
test
---
```

if: Execute Conditionally

if tests the result of a command and conditionally executes other commands based upon that result. It can be used instead of **&&** and **||**, as shown above. Instead of:

```
cmp -s file1 file2 && cat file2
```

you can use:

```
if cmp -s file1 file2
then cat file2
fi
```

This means that the shell executes

```
cat file2
```

if **cmp** returns zero (true).

To get the same result as given by the previously illustrated command:

```
cmp -s file1 file3 || rm file3
```

with the **if** statement, also use **else**:

```
if cmp -s file1 file3
then
else rm file3
fi
```

The commands between **else** and **fi** are executed if the result of the command following the **if** is false or non-zero. Note that there is no command following *then*.

The **elif** statement lets you test several conditions with one **if** statement and act on the one that is true. In general terms,

```
if command1
then action1
elif command2
then action2
elif command3
then action3
else action4
fi
```

The items labeled *command* and *action* are both commands or lists of commands.

First, the shell executes **command1**. If the result is true, it performs **action1**. If the result from **command1** is not true, the shell then executes **command2**. If its result is true, then it performs **action2**. This process continues so long as none of the commands return a true result. If none of the command results are true, the action following the **else** is executed.

To illustrate, create a shell script that list on your terminal only one of the three file-name arguments. Use the command

```
test -f name
```

which returns true if *name* is an existing non-directory file.

```
ed
a
if test -f $1
then cat $1
elif test -f $2
then cat $2
elif test -f $3
then cat $3
else echo 'None are files'
fi
.
w cat.1
q
chmod +x cat.1
```

while: Execute a Loop

Another looping or repetitive shell statement is the **while** statement. The commands

```
while command1
do command2
done
```

first performs *command1*. If its result is true, *command2* is executed, and *command1* is again executed. This process continues until *command1* returns false (non-zero).

until: Another Looping Construct

The construct **until** resembles **while**. For example, the commands:

```
until command1
do command2
done
```

execute *command2* until *command1* returns true (zero).

case: Serial Conditional Execution

The **case** statement resembles the **if** statement in that it offers a multiple choice. To illustrate, type the following script, which lets you choose one of several ways to list the contents of a directory:

```
ed
a
case $1 in
     1) ls -l;;
     2) ls;;
     3) lc;;
     *) echo unknown parameter $1;;
esac
.
w dir
q
chmod +x dir
```

The words **case** and **esac** bracket the entire **case** statement. The effect of the command

```
dir 2
```

is equivalent to:

```
ls
```

Each choice within the **case** statement is indicated by a string followed by):

```
2)
```

indicates what is to be executed if argument **$1** has the value **2**.

The strings that select the choices may be patterns. The choice '*)' signifies that a match can be made on any string. Notice that this resembles the use of * to substitute any file name. An expression of the form

```
[1-9])
```

in a **case** statement matches any digit from 1 through 9. A list of alternatives can be presented by separating the choices with a vertical bar:

```
a|b|c) command
```

Each command or command list in the case choice must be terminated by a double semicolon ;;.

Summary

The shell is a command programming language that handles simple commands as well as complex commands that can iterate as well as make decisions. Three kinds of substitution are provided to increase the power of your commands.

For more information about the shell, see the tutorial for the shell that follows in this manual. For more information about a given command, see its entry in the Lexicon.

Note, too, that the COHERENT system also includes the Korn shell **ksh**. This is a superset of the Bourne shell described here, and has many features that you may find useful. For information about this shell, see the Lexicon entry for **ksh**.

TUTORIAL

Section 14:

UUCP, Remote Communications Utility

UUCP is a set of programs that together let you communicate in an unattended manner with remote COHERENT and UNIX sites. The term *UUCP* is an abbreviation for "UNIX to UNIX copy"; as its name implies, UUCP was developed under the UNIX operating system. Mark Williams Company has recreated UUCP for COHERENT.

UUCP allows your COHERENT system to talk over telephone lines to other computers that also run COHERENT or UNIX. It can transmit files and mail to other systems and receive material from them, without needing you to guide it by hand every step of the way. Moreover, you can instruct UUCP to telephone other computers at the same time each day; this permits regular, orderly exchange of mail, news, and files among computers, and allows you to take advantage of lower telephone rates during off-peak periods. In a similar fashion, UUCP allows other systems to log into your system, to exchange mail or other information, and otherwise perform useful tasks.

Numerous UUCP systems have linked together to create an informal network called the *Usenet*. Many megabytes of source code, news, and technical information are available across the Usenet. Anyone who is connected to the Usenet can exchange mail with anyone else who is also connected to the Usenet. All that is required to hook into the Usenet is to obtain a UUCP connection to anyone else who is connected to the Usenet.

You can use UUCP only if you have telephone access to another computer that runs UUCP, and if your system and the remote system with which you wish to communicate have been described to each other. UUCP is standard with COHERENT and UNIX, and can be purchased for MS-DOS. If you wish to copy files from another system, you must arrange with the system administrator of that system before you can begin to use UUCP. Likewise, if you want someone else to dial into your system to upload or download files, you must first describe that system to your copy of UUCP.

UUCP, however, is a system that supports thousands of interlinked computers that exchange millions of bytes of data daily; as you can imagine, a subject of this scope is difficult to encompass in a document as brief as this. If you wish to explore the heights and depths of UUCP, we urge you to acquire the following books:

- O'Reilly, T.; Todino, G.: *Managing UUCP and Usenet*. Sebastopol, Calif, O'Reilly & Associates Inc., 1987.
- Seyer MD: *RS-232 Made Easy: Connecting Computers, Printers, Terminals, and Modems*. Englewood Cliffs, NJ, Prentice-Hall Inc., 1984.
- *!%@:: A Directory of Electronic Mail Addressing and Networks*. Sebastapol, Calif, O'Reilly & Associates Inc., 1989.

Contents of This Manual

This tutorial describes UUCP and tells you how to set up and run your UUCP system. It briefly describes how to attach a modem to your computer and how to describe it to the COHERENT system, and it describes how to dial out to other systems to exchange files. Future editions of this manual will contain information on how to allow other systems to dial into your system, so it can act as the hub of a network.

An Overview of UUCP

UUCP is a set of programs that exchange files with other computers that run UUCP. You can set aside files or mail messages to be transferred to another computer; UUCP regularly checks to see if material has been set aside to be transferred, dials the remote system, and exchanges the files without requiring your assistance.

This appears to be a simple function, but it can be extremely useful to you. Suppose, for example, that you run a real-estate office that is a member of an organization with regional and national offices. You can tell UUCP to call your regional office each night, to send a file of your new listings and to accept a file of new listings in your district that had come from other local offices. Likewise, your association's regional office can telephone the national office each night to receive new listings in your region, which can then be passed on automatically to the appropriate neighborhood offices. All of this information can be transferred at night, when telephone rates are lowest, and without needing you to be at the console. When you come to work the next morning, you will have the latest listings instantly available on your terminal.

In brief, what UUCP offers is the ability to join a *network* of computers, in which every user of every computer can exchange information with every user on every other computer, automatically. What computer networks can do is limited only by your need to exchange information with other computer users, and by your imagination.

The Programs

UUCP consists of the following programs:

uucp The UUCP user interface. **uucp** copies files from one computer to another. Be sure not to confuse the **uucp** command with the UUCP system, despite their similar names.

uucico Call remote systems: log in to the remote system, and transfer files.

uudecode Translate files encoded by **uuencode** back into object code.

uuencode Translate binary files into printable ASCII characters for transmission to another system.

uuinstall This program displays a template on your screen, and helps you describe a system to UUCP relatively painlessly.

uulog Read the UUCP logs, which reds the processes that UUCP has initiated recently.

uumvlog Copy the current UUCP log files into backup files, named after the day on which they were generated. Throw away all log files older a requested number of days. UUCP logs everything that it does; and since it does a lot, its log files can grow very large very quickly. **uumvlog** helps to ensure that you have enough information on your system to see where UUCP has gone wrong, yet the UUCP log files do not grow large enough to overwhelm your system.

uuname List the systems that your computer can reach.

uutouch Create a file that triggers a call to a named remote system.

uux Execute a command on a remote system.

uuxqt Execute files with the prefix "X." in the directory **/usr/spool/uucp/***sitename*.

Three other programs, while not part of UUCP per se, are used by it:

ttystat Check the status of your asynchronous ports. If UUCP is not receiving files from other systems or not sending files to other systems, it may be because the appropriate ports have not been enabled.

mail Send "electronic mail" to another person, either on your system or on another system via UUCP.

uux Execute commands on remote systems.

Directories and Files

UUCP uses the following files and directories.

/bin/uulog
The **uulog** command.

/etc/domain
This file lists the UUCP domain. It is read by **mail**.

/etc/modemcap
This file holds descriptions of modems that are understood by the COHERENT system.

/etc/uucpname
Holds the name of your system, as it is known to other UUCP sites.

/usr/bin/uucp
The **uucp** command. Copy a file to another system that runs UUCP.

/usr/bin/uuname
The **uuname** command.

/usr/bin/uudecode
The **uudecode** command.

/usr/bin/uuencode
The **uuencode** command.

/usr/lib/uucp
Contains UUCP commands and system data files.

/usr/lib/uucp/L-devices
Describe the outgoing lines. Note whether they are directly wired or modems; give the protocol needed to manipulate them.

/usr/lib/uucp/L.sys
Gives login data for remote sites. It gives the way to call remote sites and the sites that only call you.

/usr/lib/uucp/Permissions
For each site, list the programs that that site has permission to execute on your system.

/usr/lib/uucp/ttystat
The **ttystat** command.

/usr/lib/uucp/uucico
The **uucico** command.

/usr/lib/uucp/uumvlog
The **uumvlog** command.

/usr/lib/uucp/uutouch
The **uutouch** command.

/usr/lib/uucp/uuxqt
The **uuxqt** command.

/usr/spool/logs/uucp
Log of UUCP activity.

/usr/spool/uucp/.Log
Directory containing UUCP logfiles, as follows:

/usr/spool/uucp/.Log/uucico/*sitename*
/usr/spool/uucp/.Log/uux/*sitename*
/usr/spool/uucp/.Log/uucp/*sitename*
/usr/spool/uucp/.Log/uuxqt/*sitename*

/usr/spool/uucp/*sitename***/C.***
Files that instruct the local system either to send or to receive files.

/usr/spool/uucp/*sitename***/D.***
Work files for outgoing and incoming files.

/usr/spool/uucp/LCK.*
The "lock" files UUCP uses to coordinate its resources. When a UUCP program attempts to access a remote site, it writes a "lock" file for that site. This is to prevent UUCP from accidentally attempting to access the same site more than once simultaneously. When the program that wrote the lock file exits successfully, it erases its lock files, and so makes that site accessible to other UUCP programs.

/usr/spool/uucp/.Sequence
This directory contains the sequence number of the last file handled by UUCP.

/usr/spool/uucp/TM*
These are temporary files generated by uucico while receiving files from remote sites.

/usr/spool/uucp/*sitename***/X.***
Executed files. These files will be executed by the command **uuxqt**, and are generated by a remote system.

/usr/spool/uucppublic
Public directory accessible by all remote UUCP systems.

Attaching a Modem to Your Computer

It is straightforward to attach a modem to your computer.

First, read the documentation that comes with your modem, and look for the following: (1) the baud rate at which the modem operates, and (2) the command protocol that your modem uses.

TUTORIAL

Second, check the plug on the back of your modem. The modem will connect to your computer via a nine-pin or 25-pin D plug, also known as an RS-232 interface. Such a plug can be either *male* or *female*: the male plug has nine or 25 small pins projecting from it, whereas the female does not.

Third, obtain a cable to connect one of the serial ports on your computer to the modem. The serial ports on an IBM AT or AT compatible are almost always male; if your modem has a female plug, you will need a male-to-female cable, whereas if your modem's plug is male (which is very rare), you will need a female-to-female plug. Be sure to purchase a standard modem cable for an IBM AT; practically all computer dealers carry them. The cable you purchase should support "full modem control"; if it doesn't say on the package, be sure to ask your dealer before you buy it. If you are handy with a soldering iron you may be able to solder up such a cable for yourself, but unless you know precisely what you are doing it probably is not worth the trouble. File **/usr/pub/rs232** contains pinouts for both nine- and 25-pin connectors. When you plug in your cable, be sure to note whether you plugged it into port **com1**, **com2**, **com3** or **com4**.

Fourth, reconfigure the serial port to suit your modem. This involves the following steps:

1. Log in as the superuser **root**.

2. Edit the file **/etc/ttys**. This file normally has several lines in it, one that describes the console and one for each serial port. Each line has four fields: a one-character field that indicates whether a login prompt should be displayed (used only for devices from which people will be logging into your system); a one-character field that describes whether the device is local or remote (a local would be a modem from which you wished to dial out, a remote device would be a modem from which someone could dial in); a one-character field that describes the speed (or baud rate) at which the device operates; and a field of indefinite length that names the device being described. If you have plugged into serial port **com1** a 1200-baud modem that will allow remote logins, edit the line for **com1** to read as follows:

   ```
   1rIcom1r
   ```

   If you have plugged into serial port **com2** a 2400-baud modem from which you are only going to dial out, edit the line for **com2** to read as follows:

   ```
   0lLcom2l
   ```

   Note that the second and last character are a lower-case el, not a one. For more information, see the Lexicon entries for **com**, **getty** and **ttys**.

3. When you have finished editing **/etc/ttys**, type the following command:

   ```
   kill quit 1
   ```

   This will force COHERENT to read **/etc/ttys** and set up its ports in the manner that you have configured them.

Finally, test if you have connected your modem. Turn on your modem; then log in as the superuser **root** and type the following command:

```
echo "FOO" >/dev/port
```

where *port* is the "local" version of the port, depending on which serial port you have plugged your modem. If the systems are connected, the lights on your modem should blink briefly. For a more sophisticated test, try to communicate with your modem by using the command **kermit**. If you are not familiar with **kermit**, see its entry in the Lexicon for details.

If you continue to have problems making connections with your modem, see the volume *RS-232 Made Easy*, referenced above. It describes in lavish detail how to connect all manner of devices via the RS-232 interface.

Installing UUCP

Installing UUCP on your system means giving UUCP information about both your system and about the remote systems with which it will be making contact.

Before you can use UUCP to log into a remote system, you must find a remote system that will let you log in via UUCP. If you ask around among your friends or check local bulletin boards, you should have no trouble finding a UUCP system that will let you log in.

When you find a system that will let you log in, you must tell the administrator of that system what your system is called. If you have not yet selected a name for your system, do so now. The name must be eight characters or fewer, and must be unique — or unique, at least, to the system into which you will log in.

The administrator of the remote system, in turn, will give you the following information: (1) the name of his system; (2) the telephone number of his system and the speed of his modem; (3) your login name (this is apart from your system name); and (4) your password. He may also tell you when he would prefer for you to log into his system. This information should always be kept confidential, just as you would keep confidential the combination to a friend's locker.

Once you have exchanged information with the adminstrator of the remote system, you must describe that remote system to your local UUCP system. This is an involved process that has many pitfalls; however, COHERENT includes a program to make this task easier for you: **uuinstall**. This program displays a template on the screen; you fill in the blanks, and it does the rest.

To begin, log in as the superuser **root**, and type **uuinstall**. In a moment, the screen will clear and the following menu will appear:

```
H - Help for screens
S - Sitename
L - Lsys
D - Devices
P - Permissions
```

You should type **H** first. **uuinstall** will show you the keystrokes it expects to move from one field to another. These keystrokes are the same as used by the MicroEMACS screen editor; if you're not familiar with MicroEMACS, the keystrokes are easy enough to learn.

Setting Up Your Local Site

If you are describing a system to UUCP for the first time, type **S** for site name. This will ask you for two bits of information: the name you have given your site, and your local domain. Type in the site name in the space indicated. The "domain" is used to organize groups of users on your system; the Mark Williams edition of UUCP does not yet use domains, but the **mail** program expects domain information to be included for use with mail headers. Therefore, enter something into the domain slot; you may wish to use your site name followed by **.UUCP** (e.g., **mwc.UUCP**), or something similar. When you have finished entering information, type **<ctrl-Z>** to exit this screen and return to the main menu.

If you are working with UUCP for the first time, type **D** for devices. This will let you describe to UUCP the type of modem it will working with. You will see the following template:

```
Type:
Line:
Remote:
Baudrate:
brand:
```

Type **N** to go to the next entry. Keep pressing **N** until you see the entry for the communications port you wish to use. If you wish to add a device not already listed, type **A** to "add" an entry.

The first entry, **Type:** can have one of two entries: **DIR** or **ACU**. The former indicates devices that are directly wired into the computer, such as terminals; the latter is for remote devices like modems. Type **ACU**, then **<ctrl-N>** to move to the next field.

In the next field, **Line:**, enter the serial port into which you've plugged your modem: **com1l**, **com2l**, etc. Then type **<ctrl-N>**.

The next field, **Remote:**, gives the name of the port into which a remote device is connected. Enter the port into which you plugged your modem, followed by the letter 'r'. For example, if your modem is plugged into port **com2**, enter **com2r**. Type **<ctrl-N>** to move to the next field.

The next field, **Baudrate:**, is the speed at which your modem operates, e.g., 2400 or 9600. Enter it, then type **<ctrl-N>**.

Finally, enter the type of modem that you are using. The COHERENT system's file **/etc/modemcap** contains descriptions for a number of popular modems, to spare you the trouble of typing control sequences for your modem. The following table gives the code name for each of the modems described in **/etc/modemcap**, plus a description of it:

hayes	Hayes Smartmodem 1200
tbfast	Trailblazer, 9600 baud
xtb2400	Trailblazer, 2400 baud
avatex	Hayes clone, 2400 baud
promodem	Prometheus Promodem 1200
mk12	Signalman Mark XII
dc300	Radio Shack Direct-Connect 300

Enter the code name for the appropriate modem. One hint: if you have a Hayes or Hayes-compatible modem that runs at 2400 baud, enter **avatex** instead of **hayes** — their **modem** descriptions are virtually identical except for the baud rate.

Please note that the dialing commands in **modemcap** assume that you have a Touch-Tone telephone. If you have a pulse telephone, you must modify your modem's entry in **modemcap**. First, consult the documentation for your modem and find the correct command for dialing a pulse telephone; on Hayes and Hayes-compatible modems, it is **DP**. Then open the file **/etc/modemcap** and locate the description of your modem; then change the characters that follow the string **ds=** to the command you just looked up. For example, to edit the **avatex** entry in **modemcap** so it will dial a pulse telephone, change the string **ds=DT** to **ds=DP**.

If you have described your modem correctly, there should be no need for you to do it again. Type **<ctrl-Z>** to save your changes and return you to the main menu.

Describing a Remote Site

Next, type **L** for **L.sys**. **L.sys** is a file that hold a description of every system to which you will make connection. You will see the following template:

```
System
Line
baud rate
phone number

Day to Call   Time From     Time To
Any





Expect     ""
Send       ""
Expect     ogin -
Send
Expect     ssword:
Send
```

Type **M**, to *modify* **this entry.**

In the first entry, **System**, type the name of the system with which you will be connecting. For example, if the system for International Widget is named **intwidget**, type that followed by **<ctrl-N>**. If a system is described more than once in the file **L.sys**, UUCP will use the first description.

The next field, **Line**, names the line to which you have connected your modem, either **ACU** or a port from the aforementioned "devices" screen, followed by **<ctrl-N>**. This may seem redundant with the description in the **device** file; however, it's not, because it's possible to connect to a remote system via more than one route or device.

In the next field **baud rate**, enter your modem's baud rate; then **<ctrl-N>**.

In the next field, **phone number**, enter the remote system's telephone number. If the remote system has an area code other than yours, be sure to include the '1' before the telephone number; also, do not include the hyphens in the telephone number, or it will not fit into the space allotted for it. Then type **<ctrl-N>**.

Day and Time of Connection

The next of fields let you set the days of the week and times at which you wish to dial the remote system. **Day to Call** recognizes the following values:

Wk	Every weekday, i.e., Monday through Friday
Su	Sunday
Mo	Monday
Tu	Tuesday
We	Wednesday
Th	Thursday
Fr	Friday
Sa	Saturday
Never	Don't call remote system
Any	Call at any time

Time From and **Time To** set a "window" during which UUCP will attempt to contact the remote system. Both are set using a 2400-hour clock; for example, with the following setting

```
We      2100   2300
```

UUCP will try to contact this remote system between 9:00 PM and 11:00 PM. Likewise, with the following setting

```
We      2300   0200
```

UUCP will try to contact this remote system between 11:00 PM and 2:00 AM the following morning. If on the first try UUCP fails to make connection with the remote system (the line is busy, say), it will try again periodically until either it connects with the remote system or the time period for that system and day has ended. (The following section will tell you how to set when UUCP checks for newly queued files.) When the next "legal" time comes around, UUCP will then try again.

If you do not set the time for a given day, then UUCP will attempt to contact the remote system as soon as it discovers that a file for that system has been queued. The advantage of setting times is that you can force UUCP to work in the evening and on weekends, when telephone rates are cheaper, and you can spread UUCP's work around so it never overloads the system at any given point. After all, if you need your modem yourself during the day, you don't want to wait for UUCP to finish a call before you use it.

As you can see, the template for days and times has seven rows. This lets you established different times for each day of the week; for weekdays and weekends; for weekdays alone; or weekends alone. Note that you do not have to dial a remote site every day! Depending on the importance of the site, weekdays or weekends alone may be sufficient. Consider the following set of entries:

Day to Call	Time From	Time To
Wk	2300	0200
Sat	2300	0200
Sun	1300	1500

This scheme dials the remote site between 11 PM and 2 AM Monday through Saturday, and between 1 PM and 3 PM on Sunday. This takes advantage of the fact that on Sundays, lower telephone rates are in the afternoon rather than the evening; and it also takes advantage of the fact that like most sensible people, you have better things to do on a Sunday afternoon besides work on your computer.

Note that the default setting, **Any** with no times, forces UUCP to transmit files as soon as the are queued. If you wish to change this, do so; in any case, you can move from field to field and from line to line by typing **<ctrl-N>**.

If you are interacting with a number of remote sites, be sure to stagger the times during which UUCP attempts to contact them. The more systems UUCP has to contact during a given time period, the fewer attempts it will be able to make to contact any of them.

The Chat Script

The last six fields on this template

```
Expect    ""
Send      ""
Expect    ogin:
Send
Expect    ssword:
Send
```

In the UUCP world, this is called the "chat script". Basically, it walks your UUCP system through the prompts and responses by which you actually log into the remote system.

To understand the structure of the chat script, consider the process by which a user logs into the system for Universal Widget. When he makes the connection, the phrase

```
Welcome to the Wonderful World of Widgets!
```

appears on the screen. What he really wants to see, however, is the prompt

```
Login:
```

so he hits the carriage return key to demonstrate his impatience to the system. The remote system then displays the **Login:** prompt, and the user replies by typing his user ID, say "frank". Finally, the system displays the prompt

```
Password:
```

and he replies by typing his password, say "bahHumBug". All then proceeds accordingly.

The chat script is designed to imitate this sequence of events. The first **Expect/Send** pair should hold the prompt that you need to log in and how you respond if you *don't* get it. In most cases, you should set the **Expect** field to 'ogin:" and leave the **Send** field as the pair of quotation marks, which sends a carriage return.

If we were to automate the "International Widget" example above, the first **Expect/Send** pair of our chat script should read as follows:

```
Expect:   ogin:
Send:     ""
```

Note the following: First, not all systems use the word **Login:** to prompt for logins. Be sure to check with the administrator of the remote system to make sure. Second, you do not have to enter the entire prompt, only what comes at the end of it; for example, for the prompt **Login:**, the fragment **n:** is sufficient. Third, you should set these fields even if the remote system displays the login prompt immediately; the prompt may be garbled through line noise, and setting the first **Expect/Send** pair will help UUCP to cope with this.

The second **Expect/Send** pair should hold your response when you *do* receive the prompt you expect. The **Expect** half of this pair usually (but not always) holds the same prompt as the **Expect** slot of the previous **Expect/Send** pair. The **Send** half, however, should hold your login ID, as established by the administrator of the remote system. In our Widget example, the second **Expect/Send** should read:

```
Expect:   ogin:
Send:     frank
```

The third **Expect/Send** pair should hold the prompt for the password, and the password itself. In our example, the third **Expect/Send** pair should read as follows:

```
Expect:   ssword:
Send:     bahHumBug
```

When you have finished writing the chat script, your description of the remote system is complete. Type **<ctrl-Z>** to indicate that you have finished editing, and then type **X** to exit from this screen and return to the main menu. Then type **P** to enter the last template needed for installation: the one that sets permissions on your system.

The following table gives the control codes supported by COHERENT UUCP 3.0.0 and 3.1.0 in a chat script:

Notation	*Meaning*
""	Expect a null string
\b	Send backspace
\c	If send string ends with **\c**, suppress newline after send string
\d	Delay one second while sending
\n	Send a newline
\r	Send a carriage return
\s	Send a space
\t	Send a tab

The usual form of a chat script is a sequence of pairs

```
expect send
```

in which the action is to wait for the "expect" string, and on receiving it, to send the "send" string followed by a newline.

Optionally, an entry may take the form

```
expect-subsend-subexpect send
```

where **subsend** is a possibly empty string, enclosed in hyphens, to be sent if the "expect" string is *not* received. For example,

```
ogin:--ogin: nuucp
```

waits for **ogin:**; if this string is not received, a newline is sent and the system waits for **ogin:** again. If **ogin:** is received either time, **nuucp** is sent followed by a newline.

Granting Permissions

The last task in describing a remote site is setting its permissions. Unless you grant the remote system permissions, it can execute nothing on your system, not even the **mail** program to send you a letter. When you grant permissions, you name the remote system in question, name the programs it can execute on your system, name the directories into which it can write files, and the directories from which it can copy files. If permissions were not set rigidly, then every UUCP connection would be potentially a breach of system security.

uuinstall's "permissions" appears as follows:

```
Remote site name
Provide an entry for that site calling in <y/n>
Provide an entry for calling that site <y/n>
      MACHINE            LOGNAME
Commands which can be executed at this computer by this remote site:

Read directory list

Exceptions to read directory list

Write directory list

Exceptions to write directory list

Can the remote site request file transfers from this computer <y/n>
Can this computer initiate file transfers to the remote site <y/n>s
```

1 The first slot in the template asks you to name the remote site. Enter the name of the site as you entered it in the **L.sys** template. Note that **uuinstall** will automatically translate that into entries in the **MACHINE** and **LOGNAME** slots, below " by default, the **MACHINE** slot is set to the remote site's name, and the **LOGNAME** is the site name with the letter 'u' appended to the beginning.

2 The second slot in the template asks if you want to provide an entry in **/usr/lib/uucp/Permissions** for that site to call you. Enter 'y' only if that site will be dialing into your system; otherwise, enter 'n'.

3 The third slot asks if you want to provide an entry in **Permissions** for calling the remote site. If your system will be dialing into that system, enter 'y'.

4 The next slot asks you to name the commands that the remote site can execute on your computer. Enter **rmail:rnews**; the former lets the remote system send electronic mail on your system, and **rnews** lets it transfer news files to you. Add other commands if you like, but remember that the shorter the list is, the less the chance an intruder will be able to mischief on your system.

5 In the next slot, enter the directories from which the remote system can copy files. Enter **/usr/spool/uucp/uucppublic:/tmp**, plus whatever directories are appropriate.

6 The next slot requests exceptions to the read list. When you enter a directory on the read list, that directory plus all of its children become available for reading. If you wish to place "off limits" a subdirectory of any directory named in the previous slot, enter it here.

7 The next slot asks you to name the directories into which the remote system can write files. Enter **/usr/spool/uucp/uucppublic:/tmp**, plus any others that you wish.

8 Next, enter the list of exceptions to the write list.

9 The next slot asks if the remote system can ask your system to transfer files to it. If you wish to grant this permission, enter "y", which is the usual order of things.

10 Finally, **uuinstall** asks if your computer can ask the remote system to transfer files to you. If you can do so, enter 'y'.

This concludes the process of describing a remote system to UUCP. Type **<ctrl-Z>** to end data entry and return to the main menu; then enter 'X' to exit. Type 'y' when asked if you wish to save your changes into the system's files. And that's all there is to it.

Every time you wish to make contact with a new system, you must use **uuinstall** as described above.

This description may need several revisions, as you attempt to make contact with the remote system. Writing these descriptions is something of a black art. Be patient and persistant: once contact is made, the connection should work without further maintenance being needed for months to come.

Setting a Polling Time

The next step in setting up your UUCP system is to edit the file **/usr/lib/crontab**. This file contains a description of programs that are to be executed periodically. The program **cron** reads this file once every minute, checks its contents against the system time, and executes the appropriate programs. By inserting descriptions of the UUCP commands into **/usr/lib/crontab**, you will ensure that UUCP will execute regularly to poll the remote sites you have described to it. If you do not insert entries into **/usr/lib/crontab**, UUCP will connect with a remote system only if it has a file to upload to it.

The format of **/usr/lib/crontab** is described in detail in the Lexicon entry for **cron**. Basically, a crontab entry has six fields:

1. The minute in the hour when a command is to be executed (0 through 59).
2. The hour of the day when the command is to be executed (0 through 23)
3. The day of the month (1 through 31).
4. The month of the year (1 through 12).
5. The day of the week (0 through 6, with 0 indicating Sunday).
6. The command to be executed.

Fields are delimited by space characters. Note that a command can be executed more than once in any given period; just separate the multiple entries with commas. For example, if you wish to print the date and time on your terminal every 15 minutes around the clock, insert the following entry into **/usr/lib/crontab**:

```
0,15,30,45 * * * *  date >/dev/console
```

An asterisk in a field indicates that every value of the field is to be used.

The command **uutouch** forces UUCP to schedule a poll to a remote site, regardless of whether any files are waiting to be uploaded to that system. To poll the site called **george** on a regular basis, insert the following lines into **/usr/lib/crontab**:

```
30 * * * * /usr/lib/uucp/uucico -sgeorge
0 22 * * * /usr/lib/uucp/uutouch george
```

The first line invokes the program **uucico** every hour on the half hour around the clock. **uucico** checks to see if there is a file to be sent to site **george**, and dispatches it if need be.

The second line invokes the program **uutouch** every night at 10 PM. **uutouch** will schedule a poll to site **george** to see if it has a file to send to you. The next time that **uucico** is invoked, it will then call site **george**.

Finally, the command **uumvlog** should be invoked daily by **cron**. **uumvlog** copies all of UUCP's log files into backup files that are named by the date they were saved. This command takes one argument, the number of days' worth of backup files to save. For example, the command

```
uumvlog 2
```

saves two days' worth of backup files; when a backup file becomes more than two days old, **uumvlog** throws it away. UUCP is designed to log everything that it does; and since it does a great many things, the log files can grow very large, very quickly. On a small system especially, you should be ruthless in purging your UUCP log files, or you may find them overwhelming the available disk space on your system. For most users, two days' worth of log files is sufficient.

The following entry in file **/usr/lib/crontab** will purge your backup files at the stroke of midnight every day:

```
0 0 * * * su uucp sh /usr/lib/uucp/uumvlog 2
```

Sending files via UUCP

Suppose, for example that site **santa** has been described to your UUCP system, and everyone has permission to read from your current directory. Suppose, too, that you have permission to write into directory **/usr/spool/reports/parents**. To send the files **good.kids** and **bad.kids** to **santa**, type the following command:

```
uucp good.kids bad.kids santa!/usr/spool/reports/parents
```

The **uucp** command compels UUCP to copy one or more files from your site to a remote site. UUCP queues both files automatically and sends them at the next scheduled time.

Note, too, the use of the '!' in the above command. The '!' separates a site name from another site name, from a directory name, or from a user ID. In the above example, the '!' indicates that directory **/usr/spool/reports/parents** can be found at site **santa**. One feature of a UUCP network is that any member can send files to any other member. That does not mean that every member must have full permissions with every other member; rather, for the sake of efficiency it is possible to route files through one or more intermediate computers, to allow batch transmissions of files. For example, to send the file **visibility** to user **blitzen** via machines **santa** and **reindeer**, use the following command:

```
uucp visibility santa!reindeer!blitzen!/usr/spool/weather/usa
```

In this example, the string **santa!reindeer!blitzen!/usr/spool/weather** indicates that directory **/usr/spool/weather** can be contacted at site **blitzen**, which in turn can be contacted via site **reindeer**, which in turn can be contacted via site **santa**. This scenario assumes that site **reindeer** has permission to write into directory **/usr/spool/weather** on site **blitzen**, and that site **santa** has permission to upload files to site **reindeer**. (And, of course, that you have permission to upload files to site **santa**.) If any of these are not true, the transaction will fail.

With UUCP networks growing to international dimensions, such path names can become quite complex. The command **mail** has an **alias** function that allows you to define a user's UUCP path name under a simpler name that serves as that user's alias. **mail** reads the file **/usr/lib/mail/aliases** for every user listed on its command line. If it finds a match, then it substitutes the description in **aliases** for the user name. If the entry in aliases consists of two or more fields separated by exclamation points, **mail** then invokes the **uucp** command to copy the mail message to the named site. For example, if you regularly send mail to user

joe at site **widget**, then insert the following entry into **/usr/lib/mail/aliases**:

```
joe:   widget!joe
```

Make sure, first, that you have described site **widget** to UUCP or this will not work. Second, make sure that your local system does not have a user named **joe**; if it does, his mail thereafter will be shipped to the other **joe** at the remote site.

UUCP Administration

Once you have written and debugged the descriptions of your devices, systems, and permissions, administering UUCP consists mainly of reviewing the log files periodically to ensure that all connections are being made, and all programs executed correctly. The command **uulog** will assist you in this. When you type the command

```
uulog widget
```

uulog will open all of the log files associated with site **widget**, and display them for you. Given that the log files for given site are kept in four different directories, this can be a great convenience.

Logfiles are organized as follows:

/usr/spool/uucp/.Log/uucico/*sitename*
/usr/spool/uucp/.Log/uucp/*sitename*
/usr/spool/uucp/.Log/uux/*sitename*
/usr/spool/uucp/.Log/uuxqt/*sitename*

As you can see, one logfile for each site is kept in a directory named after a given UUCP command. UUCP records every transaction; so by reading these files, you can see whether your UUCP commands are succeeding.

If you are having trouble with your UUCP connections, send files through UUCP and observe how they fail. You may need to use **uuinstall** a few times to tweak your description of the remote site. If all else fails, contact Mark Williams Company.

If all is going well, you should run **/usr/lib/uucp/uumvlog** every day. This keeps the log files from getting out of hand. The previous section on setting the polling time describes how to do this.

The main task of the UUCP administrator is to monitor the UUCP log files to see that hardware is functioning correctly, and that files are transferred correctly. For example, failure to connect with a remote site after several attempts may mean that the remote site is having problems with its modem, or that it is scheduling outgoing calls for when you were scheduled to call in. Likewise, failure to receive scheduled calls from several sites may indicate equipment failure on your end. You must also monitor the alias file, to see to it that mail is routed to the correct recipient.

Finally, the UUCP administrator must monitor the use of disk space on the system. Old mail and messages, multiple copies of files, and files automatically input by various subscription and network services can eat up disk space rapidly; extraneous material must be pruned ruthlessly.

Where to Go From Here

For further information, check the Lexicon entry for each UUCP command, as well as the UUCP overview article.

Section 15:

Introduction to yacc

The first high-level programming language compiler took a very long time to write. Since then, much has been learned about how to design languages and how to translate programs written in high-level languages into machine instructions. With what is known today, the writing of a compiler takes a fraction of the time it used to require.

Much of this improvement is due to the use of more powerful software development methods. In addition, we know about the mathematical properties of computer programming languages. Software tools that apply this mathematical knowledge have played a large part in this improvement.

The COHERENT system provides two tools to simplify the generation of compilers. These tools are the lexical analyzer generator **lex** and the parser generator **yacc**. The following introduces **yacc**, and gives a basic course in its use.

Although initially intended for the development of compilers, **lex** and **yacc** have proven their utility in other, simpler, tasks. Examples of very simple languages are included in this tutorial.

yacc accepts a free-form description of a programming language and its associated parsing, and generates a C program that, when compiled, will parse a program written in the described language. It uses a left-to-right, bottom-up technique, to detect errors in the input as soon as theoretically possible. **yacc** generates parsers that handle certain grammatical ambiguities properly.

This manual presumes that you are familiar with computer-language parsing and formal methods of description of languages. Because **yacc** generates its programs in C and uses many of C's syntactic conventions, you should have a working knowledge of C. Related documents include *Using the COHERENT System* and *Introduction to lex*.

Examples

The following presents a few small examples that you can experiment with to get a feel of how to use **yacc**. Feel free to experiment with the examples to investigate new ideas.

Phrases and Parentheses

The first example describes a language we call **slang**, or *simple language*. **slang** consists of sentences. A sentence, in turn, consists of strings of letters or groups of letters enclosed in parentheses, terminated by a period. A group of letters can also include other groups of letters.

The simplest "sentence" in **slang** is:

```
a.
```

The following demonstrates a sentence that consists only of a group:

```
(ab).
```

As described above, a group can have another group inside it:

```
ab(cd(ef)).
```

The following gives the **yacc** grammar for **slang**. Type it into the file **slang.y**. Note that the lexical-analyzer routine **yylex** is included in the **yacc** input file. Note also that, as in C, comments are strings placed between the characters /* and */.

```
/* Tokens (terminals) are all caps */
%token LPAREN RPAREN OTHER PERIOD
%%
run    :      sent     /* Input can be a single */
       |      run sent /* sentence or several */
       ;
sent   :      phrase PERIOD
                     {printf ("sentence\n");}
       ;
group  :      LPAREN phrase RPAREN
                     {printf ("group\n");}
       ;

phrase :             /* empty */
       |      others
       |      group
       |      others group
       ;
others :      OTHER  /* letters and other chars */
       |      others OTHER
       ;
%%

#include <stdio.h>
#include <ctype.h>
/* Called by the parser to get a token  */
yylex ()
{
    int c;
    c = 0;

    while (c == 0) {
       c = getchar();
       if (c == '.') return (PERIOD);
       else if (c == '(') return (LPAREN);
       else if (c == ')') return (RPAREN);
       else if (c == EOF) return (EOF);
       else if (! isalpha(c)) c = 0;
    }
    return (OTHER);
}
```

To generate and compile the parser described by this input, issue the commands

```
yacc slang.y
cc y.tab.c -ly -o slang
```

Now, invoke your new parser by typing

```
slang
```

and test it by typing the following input:

```
a
a.
abc(def).
aaa(bbb(ccc)).
(a).
```

slang will reply as follows:

```
sentence
group
sentence
group
group
sentence
group
sentence
```

As you can see, **slang** recognized groups and sentences within the input you typed, and reacted by printing an appropriate message on the screen.

Simple Expression Processing

The next example creates a small language that includes two types of statements. The first type of statement resembles a procedure call, and the second is an expression. Procedure names are in upper-case letters, whereas the variables in expressions are in lower-case letters. Both procedures and expressions are terminated by a semicolon.

The following code generates a parser that identifies either the procedure being called or the arithmetic expression being calculated. The lexical input routine is independently generated by **lex**.

Enter the following program into the file **calc.y**:

```
%token VARIABLE PROCEDURE
%%
prog   :      stmnt
       |      prog stmnt
       ;
stmnt  :      stat
       |      stat '\n'
       |      error '\n'
       ;
```

```
stat    :       PROCEDURE ';'
                    {printf ("PROCEDURE is %c\n", $1);}
        |       expr ';'
                    {printf ("Expression\n");}
        ;
expr    :       expr '-' expr
                    {printf
                    ("Subtract %c from %c giving E\n",
                    $3, $1);
                    $$ = 'E';
                    }
        |       VARIABLE
                    {$$ = $1;}
        ;
```

Enter the lexical-analyzer part of the program into the file **calc.lex**:

```
%{
#include "y.tab.h"
%}
%%
[A-Z]           {
                yylval = yytext [0];
                return PROCEDURE;
                }

[a-z]           {
                yylval = yytext [0];
                return VARIABLE;
                }
\n              return ('\n');
.               return (yytext [0]);
```

Now, generate the programs and compile them by typing:

```
yacc calc.y
lex calc.lex
cc y.tab.c lex.yy.c -ly -ll -o calc
```

The following messages will appear on your console:

```
1 S/R conflict
y.tab.c:
lex.yy.c:
```

To invoke the newly generated program, type:

```
calc
```

To test it, type the following:

```
A;B;
C;
a-b-c;
a-b-c-d-e;
<ctrl-D>
```

calc will reply as follows:

```
PROCEDURE is A
PROCEDURE is B
PROCEDURE is C
Subtract c from b giving E
Subtract E from a giving E
Expression
Subtract e from d giving E
Subtract E from c giving E
Subtract E from b giving E
Subtract E from a giving E
Expression
```

Background

Now that you have tried **yacc**, the following gives some background to it, and how the parsers that it generates operate.

LR Parsing

yacc generates a "bottom up" parser. More specifically, **yacc** generates parsers that read LALR(1) languages.

LR parsers scan the input in a left-to-right fashion. Unfortunately, LR parsers for interesting languages are unpractically large. LALR(k) parsers, which are derived from LR parsers, use a "look ahead" technique, in which the next **k** elements of the input stream are used to help determine reductions. LALR(1) parsers are small enough to be practical, are easy to generate, and are fast.

Input Specification

To generate a language with **yacc**, you must specify its grammar in Backus-Naur Form (BNF). (For a good introduction to BNF, see the section on parsing in *Applied C*.) The languages recognized by **yacc**-generated parsers are rich and compare favorably with modern programming languages. The time required to generate the parser from the input grammar is very small — less than the time required to compile the generated parsers.

In addition to generating the parser to recognize the input language, **yacc** lets you include compiler actions within the grammar rules that are executed as the constructs are recognized. This greatly simplifies the entire task of writing your compiler. When used in combination with **lex**, **yacc** can make the process of writing a recognizer for a simple language the task of an afternoon.

Parser Operation

yacc generates a compilable C program that consists of a routine named **yyparse**, and the information about the grammar encoded into tables. Routines in the **yacc** library are also used.

The basic data structure used by the parser is a *stack*, or *push down list*. At any time during the parse, the stack contains information describing the state of the parse. The state of the parse is related to parts of grammar rules already recognized in the input to the parser.

At each step of the parse, the parser can take one of four actions.

The first action is to *shift*. Information about the input symbol or nonterminal is pushed onto the stack, along with the state of the parser.

The second type of action is to *reduce*. This occurs when a grammar rule is completely recognized. Items describing the component parts of the rule are removed from the stack, and the new state is pushed onto the stack. Thus, the stack is *reduced*, and the symbols corresponding to the grammar rule are *reduced* to the left part of the rule.

Third, the parser can execute an *error* action. If the current input symbol is incorrect for the state of the stack, it is not proper for the parser either to shift or reduce. As a minimum, this state will result in an error message being issued, usually

```
Syntax error
```

yacc provides capabilities for using this error state to recover gracefully from errors in the input.

Finally, the parser can *accept* the input. This means that the *start* symbol, such as *program*, has been properly recognized and that the entire input has been accepted.

Later sections discuss how you can have the parser describe its parsing actions step-by-step.

Form of yacc Programs

A **yacc** program can have up to three sections. Each section is marked by the symbol **%%**. The first section contains declarations. The second section contains the rules of the grammar. User-written routines that are to be part of the generated program can be included in the third section. The outline of **yacc** specifications is as follows:

```
definitions
%%
rules
%%
user code
```

If there are no definitions or user code, the input can be abbreviated to

```
%%
rules
```

Rules

Your language's grammar rules must be entered in a variant of BNF. The two following rules illustrate how to define an expression:

```
exp     :       VARIABLE;
exp     :       exp '-' exp;
```

Action statements that are enclosed in braces { } specify the semantics of the language, and are embedded within the rules. More information about how rules are built is given below.

Definitions

The first section in a **yacc** specification is the definitions section. This section includes information about the elements used in the **yacc** specification. Additional items are user-defined C statements, such as **include** statements, that are referenced by other statements in the generated program.

Each token, such as VARIABLE in example program **calc**, must be predefined in a **token** statement in the definitions section:

```
%token VARIABLE
```

Tokens are also called **terminals**. Only nonterminals appear as the left part of a rule, and terminals can appear only on the right side of a rule. This helps **yacc** distinguish terminals from nonterminals. Other types of statements that assist in ambiguity resolution appear here, and will be discussed in later sections.

Each grammar that **yacc** generates a parser for must have a **start** symbol. Once the start symbol has been recognized by the parser, its input is recognized and accepted. For a programming-language grammar, this nonterminal represents the entire program.

The start symbol should be declared in the definitions section as:

```
%start program
```

If no start symbol is declared, it is taken to be the left side of the first rule in the rules section.

User Code

Action statements may require other routines, such as common code-generating routines, or symbol table building routines. Such user code can be included in the generated parser after the rules section and a **%%** delimiter.

The following sections discuss definitions and rules in detail.

Rules

Rules describe how programming-language constructs are put together. Any given language can be described by many configurations of rules. This frees you to write the rules for clarity and readability.

A rule consists of a left part and a right part. The left part is said to *produce* the right part; or, the right part is said to *reduce to* the left part. A rule can also include the action the parser is to perform once it (the rule) is reduced.

General Form of Rules

Blanks and tabs are ignored within rules (except in the action parts). Comments can be enclosed between /* and */. The left part of the rule is followed by a colon. Then come the elements of the right part, followed by a semicolon.

Rules that have the same left part can be grouped together with the left part omitted and a vertical bar signifying "or". For example, the grammar

```
exp     :       VARIABLE;
exp     :       exp '-' exp;
```

can be written as:

```
exp     :       VARIABLE
        |       exp '-' exp;
```

Note that these are equivalent to the BNF:

```
<exp>   ::=     VARIABLE
<exp>   ::=     <exp> - <exp>
```

A rule can also contain C statements that are the compiler actions themselves. These actions are enclosed in braces { and } and are executed by the generated parser when the grammar rule has been recognized. More will be said about actions in the following section.

Suggested Style

Rules can be written completely free form for **yacc**. For example, the rules for the above rule can be written:

```
exp:VARIABLE|exp'-'exp;
```

However, this form is much less readable.

Two styles of **yacc** grammar are in common use. The first of these is used throughout this manual.

First, start the left part at the beginning of the line; follow it with a tab; then a colon. The right part should be on the same line, also preceded by a tab.

Second, group all rules with the same left part together, and use the vertical bar aligned under the colon for all but the first rule in the group.

Third, place action items on a separate line following the associated rule, preceded by three tabs.

Finally, precede the terminating semicolon with a single tab, to align it with the colon and vertical bar.

The outline of this style is:

```
left    :       right1 right2
                       {action1}
        |       right3 right4
                       {action2}
        ;
```

This style is compact and works well for languages whose rules and actions together are simple.

For somewhat more extensive languages, or for additional flexibility in adding statements to the action part, use the following modification of the style.

```
left    :       right1 right2 {
                       action1
                       }
        |       right3 right4 {
                       action2
                       }
        ;
```

For specifications that have larger rules or more complex actions, another style is recommended.

As in the first style, group rules with the same left part, and use the vertical bar. Place the left part, with its terminating colon, on a line by itself. Then indent the right parts of the rule one or more tabs as necessary to make the rule and actions readable. Finally, the vertical bar and the semicolon should be at the beginning of the line.

The outline for this style is as follows:

```
left:
        right1 right2 {
               action1
               }
|       right3 right4 {
               action2
               }
;
```

Since the input to **yacc** can be entirely free form, there is no restriction on how to write your rules. However, if you use a consistent style throughout, it will make your job easier.

Actions

In addition to generating a parser to recognize a specific language, **yacc** also lets you include parsing action statements. With this feature, you can include C-language action statements that will be performed when specified constructs are recognized.

Basic Action Statements

The example language **slang**, described above, the action statements simply print information on the terminal as productions are recognized:

```
sent    :       phrase PERIOD
                       {printf ("sentence\n");}
        ;
group   :       LPAREN phrase RPAREN
                       {printf ("group\n");}
        ;
```

Even if your actions will be more complex, using **printf** statements in this way can help verify your grammar early in the development process.

Action Values

If the specification is for the grammar of a programming language, the actions will normally interface to routines that access symbol tables or generate code.

yacc lets rules assume a *value* to help keep track of intermediate results within rules. These values can contain symbol-table information, code-generation information, or other semantic information.

To set a value for a rule, simply use a statement of the form

```
$$ = <value>;
```

within an action statement. The symbol **$$** is the value of the production. This value can be used by other rules that use this rule as a non-terminal part.

The example program **calc**, given above, illustrates the use of the value of productions:

```
expr    :       expr '-' expr {
                    printf
                       ("Subtract %c from %c giving E\n",
                       $3, $1);
                    $$ = 'E';
                    }
        |       VARIABLE
                       {$$ = $1;}
        ;
```

The first rule's action statement sets the value of the production **expr** to **'E'**:

```
$$ = 'E';
```

The *value* of a rule is significant in that it can be used in productions including that rule as a nonterminal part.

An example is given in the first rule above. The **printf** statement refers to the items **$1** and **$3**. **yacc** interprets these symbols to mean the value of elements one and three of the right side, respectively; that is to say, **$1** refers to the value of the first **expr** in the right side of the first rule, and **$3** refers to the second **expr**, as illustrated below:

```
expr    :       expr '-' expr
                $1   $2   $3
```

calc does not reference **$2**.

The value for the tokens is provided by the lexical analyzer. The second rule for **expr** uses this to get the value of the token **VARIABLE**. The value represented by **$1** is provided by the lexical analyzer in the statement

```
yylval = yytext [0];
```

To give another example, here is a simple calculator language, called **digit**, which performs arithemtic on one-digit numbers and prints the results. Type the following grammar into the file **digit.y**:

```
%token DIGIT
%%
session      :     calcn
       |     session calcn
       ;

calcn  :     expr '\n' /* print results */
                    {printf ("%d\n", $1);}
       ;

expr   :     term '+' term
                    {$$ = $1 + $3;}
       |     term '-' term
                    {$$ = $1 - $3;}
       ;

term   :     DIGIT
                    {$$ = $1;}
       ;

%%
#include <stdio.h>
yylex ()
{
    int c;
    c = 0;

    while (c == 0) { /* ignore control chars and space */
       c = getchar();
       if (c <= 0) return (c) /* could be EOF */;
       if (c == '\n') return (c); /* set c to ignore */

       if ((c <= '9') && (c >= '0')) {
           yylval = c - '0';
           return (DIGIT);
       }
       if (c <= ' ') c = 0;
    }
    return (c);
}
```

This creates the **yacc** specification file. To turn it into a program, type

```
yacc digit.y
cc y.tab.c -ly -o digit
```

To invoke the compiled progra, type:

```
digit
```

And to test it, type the following:

```
1+2
2+2
8+9
```

digit will reply:

```
3
4
17
```

This program is essentially an interpreter — results are calculated as numbers are typed in. When you type in

```
1+1
```

the parser recognizes the construct

```
term '+' term
```

and executes the statement that adds two numbers together. The two numbers each in turn came from the construct

```
term    :       DIGIT
```

and the value of the digit came from **yylex**. When the statement **calcn** is recognized, the value is printed as the result. Thus, the calculations are performed at the time that the constructs are recognized. If a compiler were being generated, the actions would likely build some form of intermediate code, or expression tree, as in:

```
expr    :       term '+' term
                        {$$=tree (plus, $1, $3);}
```

Structured Values

All the examples thus far have shown action values as simple **int** types. This is not sufficient for a large interpreter or compiler, because at different points in the language a value can represent a constant values, a pointer to code generation trees, or symbol table information.

To solve this problem, **yacc** allows you to define the values of **$$** and **$n** as a *union* of several types. This is done in the definitions section with the **union** statement. For example, to declare action values as an integer, tree pointer, or a symbol-table pointer, you would use the following code:

```
%union {
    int cval;
    struct tree_t tree;
    struct sytp_t sytp;
}
```

This says that action values can be a constant value **cval**, a code tree pointer **tree**, or a symbol-table pointer **sytp**.

To ensure that the correct types are used in assignments and calculations in actions in the generated C program, each token whose value will be used is declared with the appropriate type:

```
%token <tree> A B
%token <cval> CONST
```

In addition, the rules themselves can have a type declaration, as they also can pass action values. Their type is declared in the **%type** statement:

```
%type <sytp> variable
```

This declares the nonterminal **variable** to reference the **sytp** field of the value union.

The values referenced in the action statements do not need to be qualified (unless they are referencing a field of one of the union elements). **yacc** generates the necessary qualification for the references, based upon the type information provided in the **type** and **token** statements.

Keep in mind that productions that do not have explicit actions will default to an action of

```
$$ = $1
```

which might cause a type clash when compiling the generated parser. This is more likely to arise during debugging, when you have defined the types but have not put in the actions.

Handling Ambiguities

The ideal grammar for a language is readable and unambiguous. If the grammar is readable, its users will find it easy to use. If the language is unambiguous, the parser generator will parse the programs correctly. However, many common programming language constructs are ambiguous. Consider the following definition of an **if** statement:

```
statement     :     if_statement
              |     others
if_statement  :     IF cond THEN statement
              |     IF cond THEN statement ELSE statement
```

Consider a program that contains a statement

```
if a > b then if c < d then a = d else b = c;
```

The parser does not know by the grammar specification which **if_statement** the **else** belongs with. At the point of the **else**, the parser could correctly recognize it as part of the first **if** or the second **if**. The indentations illustrate the interpretation of the ambiguity associating the **else** with the first **if**.

```
if a > b then
    if c < d then
       a = d;
else
    b = c;
```

Associating it with the second **if**:

```
if a > b then
       if c < d then
              a = d;
       else
              b = c;
```

One solution to this ambiguity is to modify the language and rewrite the grammar. Some programming languages (including the COHERENT shell) have a closing element to the **if** statement, such as **fi**. The grammar for this approach is:

```
statement     :     if_statement
              |     others
if_statement  :     IF cond THEN statement FI
              |     IF cond THEN statement ELSE statement FI
```

Another ambiguity arises from a grammar for common binary arithmetic expressions. The following sample specifies binary subtraction:

```
exp    :     TERM
       |     exp '-' exp
       ;
```

For the program fragment

```
a - b - c
```

the parser can correctly interpret the expression as

```
(a - b) - c
```

or as

```
a - (b - c)
```

While for the **if** example, the language can be reasonably modified to remove the ambiguity, it is unreasonable in the case of expressions. The grammar can be rewritten for **exp** but it is less convenient.

How yacc Reacts

Because some ambiguities, such as the ones detailed above, are common, **yacc** automatically handles some of them.

The ambiguity exemplified by the **if then else** grammar is called a *shift-reduce* conflict. The parser generator can either choose to shift, meaning to add more elements to the parse stack, or to reduce, meaning to generate the smaller production. In the terms of **if**, the shift would match the **else** with the first **then**. Alternatively, the reduce choice will match the **else** with the latest (rightmost) unmatched **then**.

Unless otherwise specified, **yacc** resolves shift-reduce conflicts in favor of the shift. This means that the **if** ambiguity will be resolved in favor of matching the **else** with the rightmost unmatched **then**. Likewise, the expression

```
a - b - c
```

will be interpreted as

```
a - (b - c)
```

Additional Control

yacc provides tools to help resolve some of these ambiguities. When **yacc** detects shift-reduce conflicts, it consults the precedence and associativity of the rule and the input symbol to make a decision.

For the case of binary operators, you can define the associativity of each of the operators by use of the defining words **left** and **right**. These appear in the definition section with **token**. Any symbol appearing in **left** or **right**.

The usual interpretation of

```
a - b - c
```

is

```
(a - b) - c
```

which is called *left* associative. However, the shift/reduce conflict inherent in

```
exp '-' exp
```

is resolved in favor of the reduce, or in a right-associative manner:

```
a - (b - c)
```

To signal **yacc** that you want the left-associative interpretation, enter the grammer as:

```
%left '+' '-'
%token TERM
%%
expr   :      TERM
       |      expr '-' expr
       |      expr '+' expr
       ;
```

Some operators, such as assignment, require right associativity. The statement

```
a := b + c
```

is to be interpreted as

```
a := (b + c)
```

The **%right** keyword tells **yacc** that the following terminal is to right associate.

Precedence

Most arithmetic operators are left associative. For example, with the grammar

```
%right =
%left '-' '+' '*' '/'
%%
expr   :      expr '-' expr
       |      expr '*' expr
       |      expr '+' expr
       |      expr '/' expr
       |      expr '=' expr
       ;
```

The expression

```
a = b + c * d - e
```

based on associativity alone will be evaluated

```
a = (((b + c) * d) - e)
```

which is not according to custom. We normally think of * as having higher precedence than + or -, meaning that it is evaluated before other operators with the same associativity. The evaluation preferred is

```
a = (b + (c * d) - e)
```

To generate a parser with this evaluation, use several lines of **left**, one line for each level of precedence. Each line containing **%left** describes tokens of the same precedence. The precedence increases with each line. Thus, to get the common notion of arithmetic precedence, use a grammar of

```
%right =
%left '-' '+'
%left '*' '/'
%%
expr   :      expr '-' expr
       |      expr '*' expr
       |      expr '+' expr
       |      expr '/' expr
       |      expr '=' expr
       ;
```

This method of **%left** and **%right** gives tokens a precedence and an associativity. This can eliminate ambiguities where these operators are involved. But what about the precedence of rules or nonterminals?

To specify the precedence of rules, the **%prec** keyword at the end of the rule sets the precedence of the rule to the token following the keyword. To add unary minus to the grammar above, and to give it the precedence of multiply, use **%prec** * at the end of the unary rule.

```
%right =
%left '-' '+' '*' '/'
%%
expr    :       expr '-' expr
        |       expr '*' expr
        |       expr '+' expr
        |       expr '/' expr
        |       expr '=' expr
        |       '-' expr %prec *
        ;
```

If associativity is not specified, **yacc** will report the number of shift/reduce conflicts. When associativity is specified with **%left**, **%right** or **%nonassoc**, this is considered to reduce the number of conflicts, and thus the number of conflicts reported will not include the count of these.

Error Handling

Parsers generated by **yacc** are designed to parse correct programs. If an input program contains errors, the LALR(1) parser will detect the error as soon as is theoretically possible. The error is identified, and the programmer can correct the error and recompile.

However, in most programming environments, it is unacceptable to stop compiling after the detection of a single error. **yacc** parsers attempt to go on so that the programmer may find as many errors as possible.

When an error is detected, the parser looks for a special token in the input grammar named **error**. If none is found, the parser simply exits after issuing the message

```
Syntax error
```

If the special token **error** is present in the input grammar error recovery is modified. Upon detection of an error, the parser removes items from the stack until **error** is a legal input token and processes any action associated with this rule. **error** is the lookahead token at this point.

Processing is resumed with the token causing the error as the lookahead token. However, the parser attempts to resynchronize by reading and processing three more tokens before resuming normal processing. If any of these three are in error, they are deleted and no error message is given. Three tokens must be read without error before the parser leaves the error state.

A good place to put the **error** token is at a statement level. For example, the **calc.y** example in chapter 2 defines a statement as

```
stmnt   :       stat
        |       stat '\n'
        |       error '\n'
        ;
```

Thus, any error on a line will cause the rest of the line to be ignored.

There is still a chance for trouble, however. If the next line contains an error in the first two tokens, they will be deleted with no error message and parsing will resume somewhere in the middle of the line. To give a truly fresh start at the beginning of the line, the function **yyerrok** will cause the parser to resume normal processing immediately. Thus, an

improved grammar is

```
stmnt  :      stat
       |      stat '\n'
       |      error '\n'
                     {yyerrok;}
       ;
```

will cause normal processing to begin with the start of the next line.

Error recovery is a complex issue. This section covers only what the parser can do in recovering from syntax errors. Semantic error recovery, such as retracting emitted code, or correcting symbol table entries, is even more complex, and is not discussed here.

yacc reserves a special token **error** to aid in resynchronizing the parse. After an error is detected, the stack is readjusted, and processing cautiously resumes while three error-free tokens are processed. **yyerrok** will cause normal processing to resume immediately. The token causing the error is retained as the lookahead token unless **YYCLEARIN** is executed.

Summary

yacc is an efficient and easy-to-use program to help automate the input phase of programs that benefit by strict checking of complex input. Such programs include compilers and interactive command language processors.

yacc generates an LALR(1) parser, that implements the grammar specifying the structure of the input. A simple lexical analyser routine can be hand-constructed to fit in among the rules, or you can use the COHERENT command **lex** to generate a lexical analyzer that will fit with the parser.

As the structured input is analyzed and verified, you assign meaning to the input by writing semantic **actions** as part of the gramatical rules describing the structure of the input.

yacc parsers are capable of handling certain *ambiguities*, such as that inherent in typical **if then else** constructs. This simplifies the construction of many common grammars.

yacc provides a few simple tools to aid in error recovery. However, the area of error recovery is complex and must be approached with caution.

Helpful Hints

Until you have mastered **yacc**, the best way to build your program is to do it a piece at a time. For example, if you are writing a Pascal compiler, you might start with the grammar

```
%token PROG BEG END OTHER
program      :      PROG tokens BEG END '.'
       ;
tokens :      OTHER
       |      tokens OTHER
       ;
```

and with a simple lexical analyzer of:

```
PROGRAM             return (PROG);
BEGIN         return (BEG);
END           return (END);
.             return (yytext [0]);
```

With the generated program, you can easily test the grammar by feeding it simple programs. Then add items to both the lexical analyzer and **yacc** grammar. With this approach, you can see the parser working, and if it behaves differently than you expect, you can more easily pinpoint the cause.

If you have difficulty understanding what actions your parser is taking, **yacc** will produce for you a complete description of the generated parser. To use this, you should be familiar with the way LALR(1) parsers work. To get this verbose output, specify the **-v** option on the command line. The result will appear in the file **y.output**.

In addition, you can have the parser give you a token-by-token description of its actions while it does them, by specifying the debug option **-d**. This also generates the file **y.output**, which is helpful in reading the debug output. The debug code is generated when the **-d** option is used, but is not activated unless the **YYDEBUG** identifier is defined. Include some code in the definitions section to activate it:

```
%{
    define YYDEBUG
%}
```

Your parser can turn on and off the debugging at execution time by setting the variable **yydebug**: one for on, zero for off.

A frequent cause of grammar conflicts is the empty statement. You should use it with caution. **yacc** generates empty statements when you specify actions in the middle of a rule rather than at the end; for example:

```
def     :       DEFINE {defstart();}
                       identifier {defid ($2);}
        ;
```

yacc generates an additional rule:

```
$def    :       /* empty */
                       {defstart();}
        ;
def     :       DEFINE $def identifier {defid ($2);}
        ;
```

The resulting empty statement can cause parser conflicts if there are similar rules and the empty statement is not sufficient to distinguish between them.

Example

This tutorial closes with a larger example that incorporates most of the features of **yacc** discussed here. You can type it as shown, and modify it to improve its operation.

This example, called **nav**, calculates the great circle path and bearing from one point on the globe to another. Each pair of points is input on one line. The coordinates of the origin and destination are preceeded, respectively, by the keywords **FROM** and **TO**, and can appear in either order. Longitude and latitude are followed, respectively, by the letters **E** or **W**, and **N** and **S**. Lower-case may also be used for these letters.

The numeric part of the coordinates may be entered in degrees, minutes, and optional seconds, or in fractional degrees. You can use the symbols ^, **o**, or **d** to specify degrees because the raised circle customarily used for degrees is not available on most terminals. An apostrophe ' follows minutes, and a quotation mark " follows seconds.

As an example of using **nav**, calculate the great circle distance and initial heading from Charlestown, Indiana, to Charlestown, Australia:

```
from 38d27'n 85d40'w to 151d42'e 32d58's;
```

The result will be:

```
From lat 38.450 long 85.667 To lat -32.967 long -151.700
Distance 8030.623, Init course is 258.417
```

Here, the coordinates are echoed in decimal degrees. To exit the program, type <ctrl-D>.

To begin, type the following **yacc** specification file into the **nav.y**:

```
%{
#include "ll.h"
#define YYTNAMES
        double fromlat, fromlon, tolat, tolon;
        extern calcpath();
%}

%union {
        double dgs;
        long dgsi;
        struct ll wh;
        }

%token NEWLINE FROM TO CIRCLE QUOTE DQUOTE SEP SEMI COMMA
%token NSYM SSYM WSYM ESYM
%token <dgs> FNUM
%token <dgsi> NUM
%type <dgs> degrees long lat deg
%type <wh> where from to
%%

prob    :       single
        |       prob single
        ;

single  :       sing {
                        calcpath();
                        }
        |       error NEWLINE {
                        yyerrok; YYCLEARIN;
                        printf ("Enter line again.\n");
                        }
        ;
```

```
sing    :       from SEP to SEMI NEWLINE {
                        fromlat = $1.lat;
                        fromlon = $1.lon;
                        tolat = $3.lat;
                        tolon = $3.lon;
                        }

        |       to SEP from SEMI NEWLINE {
                        tolat = $1.lat;
                        tolon = $1.lon;
                        fromlat = $3.lat;
                        fromlon = $3.lon;
                        }

        |       to SEMI NEWLINE {
                        tolat = $1.lat;
                        tolon = $1.lon;
                        }
        ;

from    :       FROM SEP where {
                        $$ = $3;
                        }
        ;

to      :       TO SEP where {
                        $$ = $3;
                        }
        ;

where   :       lat SEP long {
                        $$.lat = $1;
                        $$.lon = $3;
                        }
        |       long SEP lat {
                        $$.lon = $1;
                        $$.lat = $3;
                        }
        ;

lat     :       degrees NSYM {
                        $$ = $1;
                        }
        |       degrees SSYM {
                        $$ = - $1;
                        }
        ;
```

```
long    :       degrees WSYM {
                        $$ = $1;
                        }
        |       degrees ESYM {
                        $$ = - $1;
                        }
        ;

degrees         :       FNUM   /* e. g. 128.3 */ {
                        $$ = $1;
                        }
        |       NUM CIRCLE NUM QUOTE /* deg min */ {
                        $$=$1 + $3/60.0;
                        }
        |       NUM CIRCLE NUM QUOTE NUM DQUOTE
                        /* and seconds */ {
                        $$=$1 + $3/60.0 + $5/3600.0;
                        }
        |       NUM CIRCLE NUM QUOTE FNUM DQUOTE {
                        $$=$1 + $3/60.0 + $5/3600.0;
                        }
        ;
%%

#include <stdio.h>
yyerror (s)

char *s;
{
    struct yytname *p;
    fprintf (stderr, "%s ", s);

    for (p = yytnames; p -> tn_name != NULL; ++p)
       if (p->tn_val == yychar) {
           fprintf (stderr, "at %s", p->tn_name);
           break;
       }
    fprintf (stderr, "\n");
}
```

Both the lexical analyzer and the parser need the following header file **ll.h**:

```
struct ll {
    double lat;
    double lon;
};
```

To turn **yacc** file **nav.y** into a program, type

```
yacc -hdr nav.tab.h -d -v nav.y
mv y.tab.c nav.y.c
```

The grammar is straightforward. The types used in the actions require a **union**, because integer degrees, floating-point degrees, and pairs of floating point degrees are used as action values. The lexical analyzer recognizes integer and floating-point numbers, and passes the value through **yylval**. The rule for **degrees** combines different degree representations to one double-precision number.

The **N**, **S**, **E**, and **W** symbols convert a location to a signed representation: **S** and **E** result in negative degrees, **N** and **W** as positive.

The rule for **where** converts the single-numbered latitude and longitude into a double number of **<wh>** type. Note that it can process the coordinates in either order.

The rule **single** handles the destination and origin in either order. It takes the pairs of coordinates from **from** and **to** and stores them in the global variables that the calculation routine uses. The error token will halt error recovery at the end of the line, so that in case of error the user can reenter the correct line. If many great circles are being computed from the same origin, you need to enter only the destination after the first time.

Once a set of coordinates has been recognized, the function **calcpath** calculates the great circle.

The error routine **yyerror** accepts an error message from the parser, and examines the table of tokens to find the name of the token where the error is detected. If it is found, it is printed. To get these token names in the program, the symbol **YYTNAMES** must be defined.

The following code gives the lexical analyzer. Type it into the file **nav.l**:

```
%{
#include "ll.h"
#include "nav.tab.h"
%}
        int integer;
        double real;
%%

[nN]            return (NSYM);
[sS]            return (SSYM);
[eE]            return (ESYM);
[wW]            return (WSYM);
o|"^"|d         return (CIRCLE);
\"              return (DQUOTE);
\'              return (QUOTE);
\n              return (NEWLINE);
from            return (FROM);
FROM            return (FROM);
to              return (TO);
TO              return (TO);
```

```
[0-9]+          {
                sscanf (yytext, "%d", &integer);
                yylval.dgsi = (long) integer;
                return (NUM);
                }

[0-9]+"."([0-9]+)? {
                sscanf (yytext, "%f", &real);
                yylval.dgs = (double) real;
                return (FNUM);
                }
,               return (COMMA);
;               return (SEMI);
[ \t]           return (SEP);
.               {
                printf ("Illegal character [%s]\n", yytext);
                return (yytext [0]);
                }
```

The lexical analyzer partitions the input into the tokens expected by the parser. For the symbols in the grammar, it returns the token type. It also recognizes integer and floating-point numbers, and converts them to integers.

Note that the **ll.h** file is required even though there is no explicit reference to its contents. This is needed because the **%union** in **nav.y** generates the header file **nav.tab.h**, referring to the **ll** structure.

Turn **lex** file **nav.l** into program by typing:

```
lex nav.l
mv lex.yy.c nav.l.c
```

Finally, you should type the following code into file **navcalc.c**. It is C code that calculates the great circle route:

```
#include <stdio.h>
#include <math.h>
/*
 *     Given latitude and longitude of start and finish,
 *     calculate the great circle path.
 */
extern double fromlon, fromlat, tolon, tolat;

calcpath ()
{
       double rad = PI / 180.0;
       double initcourse, arg, dist, d60;
       double rfromlat, rfromlon, rtolat, rtolon;
```

```
        printf ("From lat %.3f long %.3f ",
            fromlat, fromlon);
        printf ("To lat %.3f long %.3f\n",
            tolat, tolon);

        rfromlat = fromlat * rad;
        rfromlon = fromlon * rad;
        rtolat = tolat * rad;
        rtolon = tolon * rad;

        d60 = acos (
            sin (rfromlat) * sin (rtolat) +
            cos (rfromlat) * cos (rtolat) *
                cos (rfromlon - rtolon)
         );
        dist = 60 * d60 / rad;

        arg = (sin (rtolat) - cos (d60) * sin (rfromlat))
              /
              (sin (d60) * cos (rfromlat));

        initcourse = acos (arg) / rad;
        if (sin (rfromlon - rtolon) < 0)
            initcourse = 360 - initcourse;

        printf ("Distance %.3f, Init course is %.3f\n\n",
            dist, initcourse);
}
```

And now compile all three programs together.

```
cc nav.y.c nav.l.c navcalc.c -ly -lm -ll -f -o nav
```

The standard formula is used to calculate great circle path and bearing. Note that there are several limitations that are not checked for here; For example, diametrically opposite points on the globe have no unique great circle path between them. In addition, neither of the points should be at either of the poles. These checks can be added if you wish to use **nav** program as a general rather than a tutorial tool.

Where to Go From Here

The Lexicon article for **yacc** summarizes its command syntax and features. The tutorial for **lex**, the COHERENT lexical analyzer, describes how to combine **lex** with **yacc** to build applications simply.

Section 16:

The Lexicon

The rest of this manual consists of the Lexicon. The Lexicon consists of several hundred articles, each of which describes a function or command, defines a term, or otherwise gives you useful information. The articles are organized in alphabetical order.

Internally, the Lexicon has a *tree structure*. The "root" entry is the one for **Lexicon**. It, in turn, refers to a series of **Overview** entries. Each Overview entry introduces a group of entries. Each entry cross-references other entries. These cross-references point up the documentation tree, to an overview article and, ultimately, to the entry for **Lexicon** itself; down the tree to subordinate entries; and across to entries on related subjects. For example, the entry for **getchar** cross-references **STDIO**, which is its Overview article, plus **putchar** and **getc**, which are related entries of interest to the user. The Lexicon is designed so that you can trace from any one entry to any other, simply by following the chain of cross-references up and down the documentation tree.

For more information on how to use the Lexicon and how it is organized, see the entry in the Lexicon on **Lexicon**.

Example

example — Example

Give an example of Mark Williams Lexicon format
#include <example.h>
char *example(*foo*, *bar*) **int** *foo*; **long** *bar*;

This is an example of the Mark Williams Lexicon format of software documentation. At this point, each entry has a brief narration that discusses the topic in detail.

The lines in **boldface** describe how to use the function being described. The first line, **#include <example.h>**, indicates that this function requires the imaginary header file **example.h**. The second line gives the syntax of the function. **char *example** means that the imaginary function **example** returns a pointer to a **char**. *foo* and *bar* are **example**'s arguments: *foo* must be declared to be an **int**, and *bar* must be declared to be a **long**.

Example

The following program gives an example of an example.

```
main()
{
     printf("Many entries include examples\n");
}
```

See Also

Lexicon, all other related topics and functions

Notes

If a Lexicon entry uses a technical term that you do not understand, look it up in the Lexicon. In this way, you will gain a secure understanding of how to use COHERENT.

! to ~

— Preprocessing Operator

String-ize operator

The preprocessing operator # can be used within the replacement list of a function-like macro. It and its operand are replaced by a string literal, which names the sequence of preprocessing tokens that replaces the operand throughout the macro.

For example, the consider the macro:

```
#define display(x) show((long)(x), #x)
```

When the preprocessor reads the following line

```
display(abs(-5));
```

it replaces it with the following:

```
show((long)(abs(-5)), "abs(-5)");
```

Here, the preprocessor replaced **#x** with a string literal that gives the sequence of token that replaces **x**.

The following rules apply to interpreting the # operator:

1. If a sequence of white-space characters occurs within the preprocessing tokens that replace the argument, it is replaced with one space character.
2. All white-space characters that occur before the first preprocessing token and after the last preprocessing token are deleted.
3. The original spelling of the preprocessing tokens is preserved. This means that you must take care to preserve certain characters: a backslash '\' should be inserted before every quotation mark '"' that marks a string literal, and before every backslash that introduces a character constant.

Example

The following uses the operator # to display the result of several mathematics routines.

```
#include <errno.h>
#include <math.h>
#include <stdio.h>

void show(value, name)
double value, char *name;
{
     if (errno)
          perror(name);
     else
          printf("%10g %s\n", value, name);
     errno = 0;
}

#define display(x) show((double)(x), #x)
```

```
main()
{
     extern char *gets();
     double x;
     char string[64];

     for(;;) {
          printf("Enter a number: ");
          fflush(stdout);
          if(gets(string) == NULL)
               break;

          x = atof(string);
          display(x);
          display(cos(x));
          display(sin(x));
          display(tan(x));
          display(acos(cos(x)));
     }
}
```

See Also

##, #define, C preprocessor

— Preprocessing Operator

Token-pasting operator

The preprocessing operator ## can be used in both object-like and function-like macros. When used immediately before or immediately after an element in the macro's replacement list, ## joins the corresponding preprocessor token with its neighbor. This is sometimes called "token pasting".

As an example of token pasting, consider the macro:

```
#define printvar(number) printf("%s\n", variable ## number)
```

When the preprocessor reads the following line

```
printvar(5);
```

it substitutes the following code for it:

```
printf("%s\n", variable5);
```

The preprocessor throws away all white space both before and after the ## operator. This gives you an easy way to print any one of a set of strings.

must not be used as the first or last entry in a replacement list. All instances of the ## operator are resolved before further macro replacement is performed.

For more information on object-like and function-like macros, see **#define**.

See Also

#, #define, C preprocessor

Notes

Token pasting has been performed by separating the tokens to be pasted with an empty comment, but this is no longer necessary.

The order of evaluation of multiple **##** operators is unspecified.

#define — Preprocessing Directive

Define an identifier as a macro

The preprocessing directive **#define** tells the C preprocessor to regard *identifier* as a macro.

#define can define two kinds of macros: *object-like*, and *function-like*.

An object-like macro has the syntax

`#define` *identifier replacement-list*

This type of macro is also called a *manifest constant*. The preprocessor searches for *identifier* throughout the text of the translation unit, and replaces it with the elements of *replacement-list*, which is then rescanned for further macro substitutions.

For example, consider the directive:

```
#define BUFFERSIZE 75
```

When the preprocessor reads the line

```
malloc(BUFFERSIZE);
```

it replaces it with:

```
malloc(75);
```

A given *identifier* is replaced only once by a given *replacement-list*. This is to prevent such code as

```
#define FOO FOO
```

or

```
#define FOO BAR
#define BAR FOO
```

from generating an infinite loop.

A function-like macro is more complex. It has the syntax:

`#define` *identifier lparen identifier-list*$_{opt}$) *replacement-list*

The preprocessor looks for *identifier*, which is a macro that resembles a function in that it is followed by a pair of parentheses that may enclose an *identifier-list*. It replaces *identifier* with the contents of *replacement-list*, up to the first lparen '(' within *replacement-list*.

The preprocessor then examines *identifier-list* for further macros, which it expands. The modified *identifier-list* is then replaced with the rest of *replacement-list*. Pairs of parentheses that are nested between the lparen that begins *replacement-list* and the ')' that ends it are copied into *identifier-list* as literal characters. The identifiers within *identifier-list* are preserved after it has been modified by *replacement-list*. The only exceptions are identifiers that are prefixed by the preprocessing operators **#** or **##**; these are handled appropriately.

For example, the consider the macro:

```
#define display(x) show((long)(x), #x)
```

When the preprocessor reads the following line

```
display(abs(-5));
```

it replaces it with the following:

```
show((long)(abs(-5)), "abs(-5)");
```

When an argument to a function-like macro contains no preprocessing tokens, or when an argument to a function-like macro contains a preprocessing token that is identical to a preprocessing directive, the behavior is undefined.

Example

For an example of using a function-like macro in a program, see **#**.

See Also

#, ##, #undef, C preprocessor

Notes

A macro expansion always occupies exactly one line, no matter how many lines are spanned by the definition or the actual parameters. If you have defined macros that span more than one line, you must either redefine them to occupy one line, or somehow embed the newline character within the macro itself; otherwise, the macro will not expand correctly.

A macro definition can extend over more than one line, provided that a backslash '\' appears before the newline character that breaks the lines. The size of a **#define** directive is therefore limited by the maximum size of a logical source line, which can be up to at least 509 characters long.

Some implementations allowed a user to re-define a macro with a new **#define** directive. The Standard, however, allows only a "benign" redefinition; that is, the body of the new definition must exactly match the old definition, including parameter names and white space.

#elif — Preprocessing Directive

Include code conditionally

The preprocessing directive **#elif** conditionally includes code within a program. It can be used after any of the instructions **#if**, **#ifdef**, or **#ifndef**.

If the conditional expression of the preceding **#if**, **#ifdef**, or **#ifndef** directive is false (i.e., evalutates to zero) and if the current condition is true (i.e., evaluates to a value other than zero), then *group* is included within the program, up to the next **#elif**, **#else**, or **#endif** directive. An **#if**, **#ifdef**, or **#ifndef** directive may be followed by any number of **#elif** directives.

The *constant-expression* must be an integral expression, and it cannot include a **sizeof** operator, a cast, or an enumeration constant. All macro substitutions are performed upon the *constant-expression* before it is evaluated. All integer constants are treated as long objects, and are then evaluated. If *constant-expression* includes character constants, all escape sequences are converted into characters before evaluation.

See Also

#else, #endif, #if, #ifdef, #ifndef, C preprocessor

#else — Preprocessing Directive

Include code conditionally

The preprocessing directive **#else** conditionally includes code within a program. It is preceded by one of the directives **#if**, **#ifdef**, or **#ifndef**, and may also be preceded by any number of **#elif** directives. If the conditional expressions of all preceding directives evaluate to false (i.e., to zero), then the code introduced by **#else** is included within the program, up to the **#endif** directive.

A **#if**, **#ifdef**, or **#ifndef** directive can be followed by only one **#else** directive.

See Also

#elif, #endif, #if, #ifdef, #ifndef, C preprocessor

#endif — Preprocessing Directive

End conditional inclusion of code

The preprocessing directive **#endif** must follow any **#if**, **#ifdef**, or **#ifndef** directive. It may also be preceded by any number of **#elif** directives and an **#else** directive. It marks the end of a sequence of source-file statements that are included conditionally by the preprocessor.

Example

For an example of using this directive in a program, see **assert**.

See Also

#elif, #else, #if, #ifdef, #ifndef, C preprocessor

#if — Preprocessing Directive

Include code conditionally

The preprocessing directive **#if** tells the preprocessor that if *constant-expression* is true (i.e., that it evalutes to a value other than zero), then include the following lines of code within the program until it reads the next **#elif**, **#else**, or **#endif** directive.

The *constant-expression* must be an integral expression, and it cannot include a **sizeof** operator, a cast, or an enumeration constant. All macro substitutions are performed upon the *constant-expression* before it is evaluated. All integer constants are treated as long objects, and are then evaluated. If *constant-expression* includes character constants, all escape sequences are converted into characters before evaluation.

See Also

#elif, #else, #endif, #ifdef, #ifndef, C preprocessor

#ifdef — Preprocessing Directive

Include code conditionally

The preprocessing directive **#ifdef** checks whether *identifier* has been defined as a macro name. If *identifier* has been defined as a macro, then the preprocessor includes *group* within the program, up to the next **#elif**, **#else**, or **#endif** directive. If *identifier* has not been defined, however, then *group* is skipped.

An **#ifdef** directive can be followed by any number of **#elif** directives, by one **#else** directive, and must be followed by an **#endif** directive.

Example

For an example of using this directive in a program, see **assert**.

See Also

#elif, #else, #endif, #if, #ifndef, C preprocessor

#ifndef — Preprocessing Directive

Include code conditionally

The preprocessing directive **#ifndef** checks whether *identifier* has been defined as a macro name. If *identifier* has *not* been defined as a macro, then the preprocessor includes *group* within the program, up to the next **#elif**, **#else**, or **#endif** directive. If *identifier* has been defined, however, then *group* is skipped.

An **#ifndef** directive can be followed by any number of **#elif** directives, by one **#else** directive, and by one **#elif** directive.

See Also

#elif, #else, #endif, #if, #ifndef, C preprocessor, defined

#include — Preprocessing Directive

Read another file and include it
#include <*file*>
#include "*file*"

The preprocessing directive **#include** tells the preprocessor to replace the directive with the contents of *file*.

The directive can take one of two forms: either the name of the file is enclosed within angle brackets (<*header*.**h**>), or it is enclosed within quotation marks ("*header*.**h**"). Angle brackets tell **cpp** to look for *file*.**h** in the directories named with the **-I** options to the **cc** command line, and then in the standard directory. Quotation marks tell **cpp** to look for *file*.**h** in the source file's directory, then in directories named with the **-I** options, and then in the standard directory.

Most often, the file being included is a *header*, which is a file that contains function prototypes, macro definitions, and other useful material; as its name implies, it most often appears at the head of a program. The header name must be a string of characters, possibly followed by a period '.' and a single letter, usually (but not always) 'h'. A header name may have up to 12 characters to the left of the period, and names may be case sensitive.

#include directives may be nested up to at least eight deep. That is to say, a file included by an **#include** directive may use an **#include** directive to include a third file; that third file may also use a **#include** directive to include a fourth file; and so on, up to at least eight files.

Note, too, that a subordinate header file is sought relative to the original source file, rather than relative to the header that calls it directly. For example, suppose that a file **example.c** resides in directory **/v/fred/src**. If **example.c** contains the directive **#include <header1.h>**. The operating system will look for **header1.h** in the standard directory, **/usr/include**. If **header1.h** includes the directive **#include <../header2.h>** then COHERENT looks for **header2.h** not in directory **/usr**, but in directory **/v/fred**.

A **#include** directive may also take the form **#include** *string*, where *string* is a macro that expands into either of the two forms described above.

See Also

header files, C preprocessor

#line — Preprocessing Directive

Reset line number
#line *number newline*
#line *number filename newline*
#line *macros newline*

#line is a preprocessing directive that resets the line number within a file. The ANSI Standard defines the line number as being the number of newline characters read, plus one.

#line can take any of three forms. The first, **#line** *number*, resets the current line number in the source file to *number*. The second, **#line** *number filename*, resets the line number to *number* and changes the name of the file to *filename*. The third, **#line** *macros*, contains macros that have been defined by earlier preprocessing directives. When the macros have been expanded by the preprocessor, the **#line** instruction will then resemble one of the first two forms and be interpreted appropriately.

See Also

C preprocessor

Notes

Most often, **#line** is used to ensure that error messages point to the correct line in the program's source code. A program generator may use this directive to associate errors in generated C code with the original sources. For example, the program generator **yacc** uses **#line** instructions to link the C code it generates with the **yacc** code written by the programmer.

#undef — Preprocessing Directive

Undefine a macro
#undef *identifier*

The preprocessing directive **#undef** tells the C preprocessor to disregard *identifier* as a macro. It undoes the effect of the **#define** directive.

See Also

#define, C preprocessor

__DATE__ — Macro

Date of translation

__DATE__ is a preprocessor constant that is defined by the C preprocessor. It represents the date that the source file was translated. It is a string literal of the form

```
"Mmm dd yyyy"
```

where **Mmm** is the same three-letter abbreviation for the month as is used by **asctime**; **dd** is the day of the month, with the first **d** being a space if translation occurs on the first through the ninth day of the month; and **yyyy** is the current year.

The value of **__DATE__** remains constant throughout the processing of the a module of source code. It may not be the subject of a **#define** or **#undef** preprocessing directive.

Example

The following prints the preprocessor constants set by COHERENT:

```
main()
{
	printf("Date: %s\n", __DATE__);
	printf("Time: %s\n", __TIME__);
	printf("File: %s\n", __FILE__);
	printf("Line No.: %d\n", __LINE__);
	printf("ANSI C? %s\n", __STDC__ ? "Yes" : "No");
}
```

See Also

__FILE__, __LINE__, __STDC__, __TIME__, C preprocessor

__FILE__ — Macro

Source file name

__FILE__ is a preprocessor constant that is defined by the C preprocessor. It represents, as a string constant, the name of the current source file being translated.

__FILE__ may not be the subject of a **#define** or **#undef** preprocessing directive, but it may be altered with the **#line** preprocessing directive.

Example

For an example of how to use **__FILE__** in a program, see **__DATE__**.

See Also

__DATE__, __LINE__, __STDC__, __TIME__, C preprocessor

__LINE__ — Macro

Current line within a source file

__LINE__ is a preprocessor constant that is defined by the C preprocessor. It represents the current line within the source file. The ANSI Standard defines the current line as being the number of newline characters read, plus one.

__LINE__ may not be the subject of a **#define** or **#undef** preprocessing directive.

Example

For an example of how to use **__LINE__** in a program, see **__DATE__**.

See Also

__DATE__, __FILE__, __STDC__, __TIME__, C preprocessor

__STDC__ — Macro

Mark a conforming translator

__STDC__ is a preprocessor constant that is defined by the C preprocessor. If it is defined to be equal to one, then it indicates that the translator conforms to the ANSI Standard.

The value of **__STDC__** remains constant throughout the entire program, no matter how many source files it comprises. It may not be the subject of a **#define** or **#undef** preprocessing directive.

Example

For an example of using **__STDC__** in a program, see **__DATE__**.

See Also

__DATE__, __FILE__, __LINE__, __TIME__, C preprocessor

__TIME__ — Macro

Time source file is translated

__TIME__ is a preprocessor constant that is defined by the C preprocessor. It represents the time that a source file is translated. It is a string literal of the form:

```
"hh:mm:ss"
```

This is the same format used by the function **asctime**.

The value of this preprocessor constant remains constant throughout the processing of the translation unit. It may not be the subject of a **#define** or **#undef** preprocessing directive.

Example

For an example of how to use **__TIME__** in a program, see **__DATE__**.

See Also

__DATE__, __FILE__, __LINE__, __STDC__, C preprocessor

_exit() — COHERENT System Call (libc)

Terminate a program
void _exit(*status***) int** *status***;**

_exit terminates a program directly. It returns *status* to the calling program, and exits. Unlike the library function **exit**, **_exit** does not perform extra termination cleanup, such as flushing buffered files and closing open files.

_exit should be used only in situations where you do *not* want buffers flushed or files closed. For example, you may wish to call **_exit** if your program detects an irreparable error condition and you want to "bail out" to keep your data files from being corrupted.

_exit should also be used with programs that do not use STDIO. Unlike **exit**, **_exit** does not use STDIO. This will help you create programs that are extremely small when compiled.

See Also

close(), COHERENT system calls, wait()

Notes

If a program leaves **main()** by an error condition, contents of register AX becomes the exit code. Usually, these register contents are random. If you want to test a program's return code, you must to exit or return from **main()**.

A

abort() — General Function (libc)

End program immediately
void abort()

abort terminates a process with a core dump, creating a file called **core**, and prints a message on the screen. It is normally invoked in situations that "should not happen". For example, **malloc** invokes **abort** if it discovers a corrupt storage arena.

Where possible, **abort** executes a machine instruction that causes the processor to trap. If the signal associated with the trap is caught or ignored, the dump will not be produced.

See Also

_exit(), core, exit(), general functions

abs() — General Function (libc)

Return the absolute value of an integer
int abs(*n*) int *n*;

abs returns the absolute value of integer *n*. The *absolute value* of a number is its distance from zero. This is *n* if *n*>=0, and -*n* otherwise.

Example

This example prompts for a number, and returns its absolute value.

```
#include <ctype.h>
#include <stdio.h>

main()
{
     extern char *gets();
     extern int atoi();
     char string[64];
     int counter;
     int input;

     printf("Enter an integer: ");
     fflush(stdout);
     gets(string);

     for (counter=0; counter < strlen(string); counter++) {
          input = string[counter];

          if (!isascii(input)) {
               fprintf(stderr,
                    "%s is not ASCII\n", string);
               exit(1);
          }
```

```
            if (!isdigit(input))
                  if (input != '-' || counter != 0) {
                        fprintf(stderr,
                              "%s is not a number\n", string);
                        exit(1);
                  }
      }

      input = atoi(string);
      printf("abs(%d) is %d\n", input, abs(input));
      exit(0);
}
```

See Also

fabs(), floor(), general functions, int

Notes

On two's complement machines, the **abs** of the most negative integer is itself.

ac — Command

Summarize login accounting information
ac [**-dp**] [**-w** *wfile*][*username* ...]

One of the accounting mechanisms available on the COHERENT system is login accounting, which keeps track of the time each user spends logged into the system. Login accounting is enabled by creating the file **/usr/adm/wtmp**. Thereafter, the routines **date**, **login**, and **init** write raw accounting data to **/usr/adm/wtmp** to record the time, the name of the terminal, and the name of the user for each date change, login, logout, or system reboot.

The command **ac** summarizes the raw accounting data. By default, **ac** prints the total connect time found in **/usr/adm/wtmp**. Any *username* restricts the summary to each specified user.

The following options are available:

-d Itemize the output into daily (midnight to midnight) periods.

-p Print individual totals.

-w Use *wfile* rather than **/usr/adm/wtmp** as the raw data file.

Files

/usr/adm/wtmp

See Also

commands, date, init, login, sa, utmp.h

Notes

The file **/usr/adm/wtmp** can become very large; therefore, it should be truncated periodically. Special care should be taken if login accounting is enabled on a COHERENT system with a small disk.

access() — COHERENT System Call (libc)

Check if a file can be accessed in a given mode
#include <access.h>
int access(*filename, mode***) char ***filename***; int** *mode***;**

access checks whether a file or directory can be accessed in the mode you wish. *filename* is the full path name of the file or directory you wish to check. *mode* is the mode in which you wish to access *filename*, as follows:

AREAD	Read a file
AEXISTS	Check if a file exists
ALIST	List the contents of a directory
AWRITE	Write into a file
ADEL	Delete files from a directory
AEXEC	Execute a file
ASRCH	Search a directory
AAPPND	Append to a file
ACREAT	Create a file in a directory

The header file **access.h** defines these values, which may be logically combined to produce the *mode* argument.

If *mode* is **AEXISTS**, **access** tests only whether *filename* exists, and whether you have permission to search all directories that lead to it.

access returns zero if *filename* can be accessed in the requested mode, and a nonzero value if it cannot. Note that the return value is the opposite of the intuitive value, i.e., zero means success rather than failure.

access uses the *real* user id and *real* group id (rather than the *effective* user id and *effective* group id), so set user id programs can use it.

Example

The following example checks if a file can be accessed in a particular manner.

```
#include <access.h>
#include <stdio.h>

main(argc, argv)
int argc; char *argv[];
{
	int mode;
	extern int access();

	if (argc != 3) {
		fprintf(stderr, "Usage: access filename mode\n");
		exit(1);
	}
```

```
        switch (*argv[2]) {
        case 'x':
             mode = AEXEC;
             break;

        case 'w':
             mode = AWRITE;
             break;

        case 'r':
             mode = AREAD;
             break;

        default:
             fprintf(stderr,
                  "modes: x = execute, w = write, r = read\n");
             exit(1);
             break;
        }

        if (access(argv[1], mode)) {
             printf("file %s not found in mode %d\n", argv[1], mode);
             exit(0);
        } else
             printf("file %s accessible in mode %d\n",
                       argv[1], mode);
        exit(0);
}
```

See Also

access.h, COHERENT system calls, path()

Notes

When the superuser **root** executes **access**, it always returns readable/writable/executable for any file that exists, regardless of permissions.

access.h — Header File

Check accessibility
#include <access.h>

The header file **access.h** declares the function **access** and a set of associated manifest constants. You can use these to check the accessibility of a given file.

See Also

access, header files

acct() — COHERENT System Call (libc)

Enable/disable process accounting
acct(*file***)**
char ****file*;**

Process accounting records who initiates each system process and how long each process takes to execute. These data can be analyzed, to administer the system most efficiently.

The system call **acct** enables or disables process accounting. If *file* is not NULL, then accounting is turned on; if *file* is NULL, however, then process accounting is turned off.

It is usual, but not necessary, that *file* be **/usr/adm/acct**. *file* must exist. When enabled, the system appends a raw accounting data record in the format described by **acct.h** to *file* as each process terminates.

acct is restricted to the superuser.

See Also

ac, acct.h, accton, COHERENT system calls, exit(), sa, times()

Diagnostics

Successful calls return zero. **acct** returns -1 for errors, such as nonexistent *file* or invocation by a user other than the superuser.

Notes

The system writes accounting records for a process only when the process exits. Processes that never terminate and processes running at the time of a system crash do not produce accounting information.

acct.h — Header File

Format for process-accounting file
#include <acct.h>

Process accounting is a feature of the COHERENT system that allows it record what processes each user executes and how long each process takes. These data can be used to track how much each user uses the system.

The function **acct** turns process accounting off or on. When process accounting has been turned on, the COHERENT system writes raw process-accounting information into an accounting file as each process terminates. Each entry in the accounting file, normally **/usr/adm/acct**, has the following form, as defined in the header file **acct.h**:

```
struct acct {
        char        ac_comm[10];
        comp_t      ac_utime;
        comp_t      ac_stime;
        comp_t      ac_etime;
        time_t      ac_btime;
        short       ac_uid;
        short       ac_gid;
        short       ac_mem;
        comp_t      ac_io;
        dev_t       ac_tty;
        char        ac_flag;
};

/* Bits from ac_flag */
#define   AFORK     01     /* has done fork, but not exec */
#define   ASU       02     /* has used superuser privileges */
```

Every time a process performs an **exec** call, the contents of **ac_comm** are replaced with the first ten characters of the file name. The fields **ac_utime** and **ac_stime** represent the CPU time used in the user program and in the system, respectively. **ac_etime** represents the elapsed time since the process started running, whereas **ac_btime** is the time the process

started. The effective user id and effective group id are **ac_uid** and **ac_gid. ac_mem** gives the average memory usage of the process. **ac_io** gives the number of blocks of input-output. **ac_tty** gives the controlling typewriter device major and minor numbers.

For some of the above times, the **acct** structure uses the special representation **comp_t,** defined in the header file **types.h.** It is a floating point representation with three bits of base-8 exponent and 13 bits of fraction, so it fits in a **short** integer.

See Also

acct(), accton, header files, sa

accton — Command

Enable/disable process accounting
/etc/accton [*file*]

One of the accounting mechanisms available on the COHERENT system is *process accounting*, also called *shell accounting*. Process accounting records each process, who initiates it, and how long it takes to execute.

By issuing the command **accton** with a *file* argument, you enable process accounting. The system then writes raw accounting data into *file*, which normally should be **/usr/adm/acct**. The command **sa** summarizes the resulting raw statistics. If issued with no argument, **accton** disables process accounting.

accton is normally invoked by the initialization command file **/etc/rc**.

Files

/usr/adm/acct — Raw accounting data

See Also

ac, acct, acct.h, commands, init, sa

Notes

As the file **/usr/adm/acct** can become very large, special care should be taken if process accounting is enabled on a COHERENT system with a small disk file system.

acos() — Mathematics Function (libm)

Calculate inverse cosine
#include <math.h>
double acos(*arg***) double** *arg***;**

acos calculates the inverse cosine. *arg* should be in the range of -1.0, 1.0. It returns the result, which is in the range of from zero to radians.

Example

This example demonstrates the mathematics functions **acos, asin, atan, atan2, cabs, cos, hypot, sin,** and **tan.**

```
#include <math.h>
#define display(x) dodisplay((double)(x), #x)

dodisplay(value, name)
double value; char *name;
```

```
{
     if (errno)
          perror(name);
     else
          printf("%10g %s\n", value, name);
     errno = 0;
}

main()
{
     extern char *gets();
     double x;
     char string[64];

     for(;;) {
          printf("Enter number: ");
          if(gets(string) == NULL)
               break;

          x = atof(string);
          display(x);
          display(cos(x));
          display(sin(x));
          display(tan(x));
          display(acos(cos(x)));

          display(asin(sin(x)));
          display(atan(tan(x)));
          display(atan2(sin(x),cos(x)));
          display(hypot(sin(x),cos(x)));
          display(cabs(sin(x),cos(x)));
     }
}
```

See Also

errno, errno.h, mathematics library, perror()

action.h — Header File

Describe parsing action and goto tables
#include <action.h>

action.h is a header that defines structures and manifest constants used in parsing and goto tables.

See Also

header files

address — Definition

An **address** is the location where an item of data is stored in memory.

On the i8086, a physical address is a 20-bit number. The i8086 builds an address by left-shifting a 16-bit segment address by four bits, and then adding it to a 16-bit offset address. The segment address points to a particular chunk of memory. The i8086 uses four segment

registers, each of which governs a different portion of a program, as follows:

CS	Address of code segment
DS	Address of data segment
ES	Address of "extra" segment
SS	Address of stack segment

SMALL-model programs use only the offset address; hence, their pointers are only 16 bits long, equivalent to an **int**. LARGE-model programs use both segment and offset addresses. Their addresses are 20 bits long, which must be stored in a 32-bit pointer, equivalent to a **long**. COHERENT currently supports SMALL model.

On the M68000, an address is simply a 24-bit integer that is stored as a 32-bit integer. The upper eight bits are ignored; this is not true with the more advanced microprocessors in this family, such as the M68020. The M68000 uses no segmentation; memory is organized as a "flat address space", with no restrictions set on the size of code or data.

On machines with memory-mapped I/O, such as the 68000, some addresses may be used to control or communicate with peripheral devices.

Example

The following printes the address and contents of a given byte of memory.

```
#include <stdio.h>

main()
{
     char byte = 'a';
     printf("Address == %x\tContents == \"%c\"\n",
          &byte, byte);
}
```

See Also

data formats, definitions, pointer

Notes

COHERENT is in i8086 SMALL model; thus, its addresses are 16 bits long.

aha154x — Device Driver

Adaptec AHA-154x device driver

The device driver **aha154x** lets you use SCSI interface devices attached to an Adaptec AHA-154x series host adapter. This driver has major number 13. It can be accessed either as a block-special device or as a character-special device. The minor number specifies the device and partition number for disk-type devices, letting you use up to eight SCSI-IDs, with up to four logical unit numbers (LUNs) per SCSI-ID and up to four partitions per LUN.

The first **open** call on a SCSI disk device allocates memory for the partition table and reads it into memory.

Controller Configuration

Prior to installing the Adaptec host adapter in your system, you must configure the I/O base address, interrupt vector and DMA channel as follows:

I/O base address:	0x330
DMA channel:	5
Interrupt vector:	IRQ11

In addition, if you are using any synchronous SCSI peripherals, disable the synchronous transfer option on the Adaptec host adapter.

After verifying that your controller works with COHERENT, you may select an alternate I/O base address or an alternate interrupt vector. Device driver variables **SDBASE_** and **SDIRQ_** correspond to the I/O base address and interrupt vector, respectively. See Lexicon article **hs** for an example of how to configure a device driver.

When processing BIOS I/O requests prior to booting COHERENT, the Adaptec host adapter uses translation mode drive parameters: number of heads, cylinders, and sectors per track. Most current versions of the AHA-154x use values of 64 heads and 32 sectors per track, and calculate the number of cylinders based upon drive capacity. Note that these numbers are called "translation-mode" parameters because they have nothing to do with the geometry of the physical drive. Some early versions of the AHA-154x, and some versions distributed by Tandy, use 16 heads and 32 sectors per track. Device driver variable **SD_HDS_** is initialized to 64 as shipped; it should be patched to a value of 16 for adapters whose BIOS code uses 16-head translation mode. The translation-mode parameters used by the BIOS code present on your host adapter can be obtained using the **dpb** utility found on the boot diskette of versions 3.2.0 and later of COHERENT. Note that the BIOS code is executed by COHERENT only during initial bootstrap. After that, drive parameters are of no consequence since SCSI I/O requests are based upon logical block number, rather than on cylinder/head/sector addressing.

The installation procedure for COHERENT versions 3.2.0 and later patches all necessary variables for the accompanying version of the **aha154x** driver by executing the command:

```
/etc/mkdev scsi
```

Minor Device Numbers

The minor device number is decoded as follows:

Bit number:	7 6 5 4 3 2 1 0
Meaning:	S I I I L L P P

where *S* indicates the "special" bit, *III* indicates a three-bit field containing the SCSI-ID in the range of zero through seven, *LL* indicates a two-bit field containing a LUN in the range of zero through three, and *PP* indicates a two-bit field that contains either a partition number for disk-type devices or a set of special modes for devices other than disks.

The "special" bit and the partition number interact as follows:

Description	*S Bit*	*PP*	*Device*	*Type*
partition a	0	00	/dev/sd?a	disk
partition b	0	01	/dev/sd?b	disk
partition c	0	10	/dev/sd?c	disk
partition d	0	11	/dev/sd?d	disk
partition table	1	00	/dev/sd?x	disk
no rewind	1	01	/dev/sd?n	tape
RESERVED	1	10	---	----
rewind on close	1	11	/dev/sd?	tape

Loading the Driver

The **aha154x** loadable device driver must be loaded on a system that does not have a SCSI hard disk as the root device. To do so, use the command **/etc/drvld**, as follows:

```
/etc/drvld -r /drv/aha154x
```

Files

/dev/sd* — block-special devices
/dev/rsd* — character-special devices

See Also

device drivers, drvld, scsi

Notes

This release of the **aha154x** device driver only supports disk-type devices. A future version of the driver will add support for tape-type and other devices.

alarm() — COHERENT System Call (libc)

Set a timer
alarm(*n*)
unsigned *n*;

alarm sets a timer associated with the requesting process to go off in *n* seconds. After *n* seconds, the system sends the signal **SIGALRM** to the process. An argument of zero turns off the alarm timer.

By default, the receipt of the **SIGALRM** signal terminates the process. However, it may be caught or ignored by using **signal**. Because of scheduling variation and the one second granularity, the action of **alarm** is predictable only to within one second.

alarm is useful for such things as timeouts. For example, the login process on a dial-in port might hang up the line after a sufficient time has elapsed with no user response.

alarm returns the previous alarm value, which represents the time remaining from the previous call. Time remaining is superseded by the new alarm value.

See Also

COHERENT system calls, signal(), sleep()

alias — Command

Set an alias
alias [*name*[=*value* ...]]

The command **alias** is used by the Korn shell **ksh** to set or display an alias.

When called without an argument, **alias** lists all aliases that have been set so far. When called with a *name* argument alone, it lists alias of *name*, assuming one has been set.

When called with one or more arguments of the form *name=value*, it established *name* as an alias for the command *value*. For example, the command

```
alias FOO="echo bar"
```

establishes the string **FOO** as an alias for the command **echo bar**. Thereafter, when you type **FOO** on the shell's command line, it will execute the command **echo bar** and so echo the string **bar** on your terminal.

The Korn shell sets a number of aliases by default. See the Lexicon entry for **ksh** for a list of these aliases and their settings.

See Also

commands, ksh, unalias

aliases — Technical Information

File of users' aliases
/usr/lib/mail/aliases
$HOME/.aliases
$HOME/.forward

aliases is a file that holds aliases by which users on your system and other systems are known. An "alias", in effect, gives another name by which you can address a mail message to a user on either your or another system. It can serve as a mnemomic, a "mailing list", or to spare you the trouble of typing a complicated UUCP path name.

The format of each alias is

```
alias_name:     target
```

where *alias_name* gives the alias to which you mail your message, and *target* is the place where **smail** actually directs the message. *target* can be a login identifier on your local system; a mail address of a user on another system, or a cluster of users on your system, on remote systems, or both.

smail ignores differences in case when it compares a name with an alias. Lines that start with a white-space character continue from the previous line. **smail** ignores strings within parentheses, as well as any text that appears after a pound sign '#'.

Prior to delivering local mail, **smail** checks file **$HOME/.forward** for forwarding instructions. This feature can be used to forward inbound mail for a user to another machine or even a group of machines.

Examples

The following gives an example form of **aliases**:

```
# this whole line is a comment

# "mail programmers" sends mail to local users joe, jack, and bill
programmers:        joe jack bill

# same as above
programmers:        joe jack
                    bill

# same as above
programmers         joe jack
                    bill

# same as above
programmers         joe   # Joe Smith
                    jack  # Jack Thomas
                    bill  # Bill Williams
```

```
# and yet another way; note use of parentheses to comment text
programmers          joe (Joe Smith) jack (Jack Thomas)
                     bill (Bill Williams)

# send a message to someone on another system
joe:                 boston!widget!js

# send a message to users on both your and another system
programmers:         boston!widget!js    # Joe Smith
                     chicago!gadget!jt   # Jack Thomas
                     bill                # Bill Williams

# all members of "programmers" group work at site "widget"
programmers!widget          joe jack bill
```

Mailing lists are easily handled by two forms of file inclusion. The first form is the same as is supported by **sendmail**:

```
fredlist             :include:/usr/lib/mail/fredlist
```

smail adds each entry in **/usr/lib/mail/fredlist** to the alias for **fredlist**.

The second form allows **/usr/lib/mail/aliases** to include other aliases files:

```
:include:/usr/lib/mail/morealiases
```

This adds the contents of **/usr/lib/mail/morealiases** to those of **/usr/lib/mail/aliases** as a regular alias file.

All aliases are recursive, so you must be careful when defining them. For example, the entries

```
bill:        joe
joe:         bill
```

causes an infinite loop. **smail** attempts to prevent infinite loops, and to guess what you intended to do. The following example illustrates how an alias can be used to deliver mail to a remote user as well as to a local user having the same name as the alias being expanded. **smail** expands the alias

```
mylogin:             mypc!mylogin  mylogin
```

to

```
mypc!mylogin mylogin
```

even though the second occurrence of **mylogin** matches the alias name.

Both forms of file inclusion are recursive, too, and may lead to infinite loops if handled carelessly.

See Also

mail, technical information

alignment — Definition

Alignment refers to the fact that some microprocessors require the address of a data entity to be *aligned* to a numeric boundary in memory so that *address* modulo *number* equals zero. For example, the M68000 and the PDP-11 require that an integer be aligned along an even address, i.e., *address***%2==0**.

Generally speaking, alignment is a problem only if you write programs in assembly language. For C programs, COHERENT ensures that data types are aligned properly under foreseeable conditions. You should, however, beware of copying structures and of casting a pointer to **char** to a pointer to a **struct**, for these could trigger alignment problems.

Processors react differently to an alignment problem. On the VAX or the i8086, it causes a program to run more slowly, whereas on the M68000 it causes a bus error.

See Also

data types, definitions

alloc.h — Header File

Define the allocator
#include <sys/alloc.h>

alloc.h defines manifest constants and structures that are used internally with memory allocation.

See Also

header files

ar — Command

The librarian/archiver
ar *option* [*modifier*][*position*] *archive* [*member* ...]

The librarian **ar** edits and examines libraries. It combines several files into a file called an *archive* or *library*. Archives reduce the size of directories and allow many files to be handled as a single unit. The principal use of archives is for libraries of object files. The linker **ld** understands the archive format, and can search libraries of object files to resolve undefined references in a program.

The mandatory *option* argument consists of one of the following command keys:

d Delete each given *member* from *archive*. The **ranlib** header is updated if present.

m Move each given *member* within *archive*. If no *modifier* is given, move each *member* to the end. The ranlib header is modified if present.

p Print each *member*. This is useful only with archives of text files.

q Quick append: append each *member* to the end of *archive* unconditionally. The ranlib header is not updated.

r Replace each *member* of *archive*. If *archive* does not exist, create it. The optional *modifier* specifies how to perform the replacement, as described below. The ranlib header is modified if present.

t Print a table of contents that lists each *member* specified. If none is given, list all in *archive*. The modifier **v** tells **ar** to give you additional information.

x Extract each given *member* and place it into the current directory. If none is specified, extract all members. *archive* is not changed.

The *modifier* may be one of the following. The modifiers **a**, **b**, **i**, and **u** may be used only with the **m** and **r** options.

a If *member* does not exist in *archive*, insert it after the member named by the given *position*.

b If *member* does not exist in *archive*, insert it before the member named by the given *position*.

c Suppress the message normally printed when **ar** creates an archive.

i If *member* does not exist in *archive*, insert it before the member named by the given *position*. This is the same as the **b** modifier, described above.

k Preserve the modify time of a file. This modifier is useful only with the **r**, **q**, and **x** options.

s Modify an archive's ranlib header, or create it if it does not exist. This is used only with the **r**, **m**, and **d** options.

u Update *archive* only if *member* is newer than the version in the *archive*.

v Generate verbose messages.

All archives are written into a specialized file format. Each archive starts with a "magic number" called **ARMAG**, which identifies the file as an archive. The members of the archive follow the magic number; each is preceded by an **ar_hdr** structure. For information on this structure, see **ar.h**. The structure is followed the data of the file, which occupy the number of bytes specified by the variable **ar_size**.

See Also

ar.h, commands, ld, nm, ranlib

Notes

It is recommended that each object-file library you create with **ar** have a name that begins with the string **lib**. This will allow you to call that library with the **-l** option to the **cc** command.

ar.h — Header File

Format for archive files
#include <ar.h>

An *archive* is a file that has been built from a number of files. Archives are maintained by the **ar** command. Usually, an archive is a library of object files used by the linker **ld**.

The header **ar.h** describes the format of an archive. All archives start with a magic number **ARMAG**, which identifies the file as an archive. The members of the archive follow the magic number, each preceded by the structure **ar_hdr**:

```
#define DIRSIZ   14              /* from <dir.h> */
#define ARMAG    0177535         /* magic number */
```

```
struct ar_hdr {
        char    ar_name[DIRSIZ]; /* member name */
        time_t  ar_date;         /* time inserted */
        short   ar_gid;          /* group owner */
        short   ar_uid;          /* user owner */
        short   ar_mode;         /* file mode */
        size_t  ar_size;         /* file size */
};
```

The structure at the head of each member is immediately followed by **ar_size** bytes, which are the data of the file.

To enhance the performance of **ld**, the command **ranlib** provides a random library facility. **ranlib** produces archives that contain a special entry named **__.SYMDEF** at the beginning.

All integer members of the structure (everything but **ar_name**) are in canonical form to ease portability. See **canon.h** for more information.

See Also

ar, canon.h, header files, ld, ranlib

arena — Definition

An **arena** is the area of memory that is available for a program to allocate dynamically at run time. It is divided into *allocated* and *unallocated* blocks. The unallocated blocks together form the "free memory pool".

Portions of the arena can be allocated using the functions **malloc**, **calloc**, or **realloc**; returned to the free memory pool with **free**; or checked to see if they are allocated or not with **notmem**. To check whether the arena has been corrupted or not, use the function **memok**.

See Also

calloc(), definitions, free(), malloc(), memok(), notmem(), realloc()

argc — C Language

Argument passed to main()
int argc;

argc is an abbreviation for "argument count". It is the traditional name for the first argument to a C program's **main** routine. By convention, it holds the number of arguments that are passed to **main** in the argument vector **argv**. Because **argv[0]** is always the name of the command, the value of *argc* is always one greater than the number of command-line arguments that the user enters.

Example

For an example of how to use **argc**, see the entry for **argv**.

See Also

argv, C language, envp, main()

argv — C Language

Argument passed to main()
char *argv[];

argv is an abbreviation for "argument vector". It is the traditional name for a pointer to an array of string pointers passed to a C program's **main** function; by convention, it is the second argument passed to **main**. By convention, **argv[0]** always points to the name of the command itself.

Example

This example demonstrates both **argc** and **argv[]**, to recreate the command **echo**.

```
main(argc, argv)
int argc; char *argv[];
{
     int i;

     for (i = 1; i < argc; ) {
          printf("%s", argv[i]);
          if (++i < argc)
               putchar(' ');
     }

     putchar('\n');
     exit(0);
}
```

See Also

argc, C language, envp, main()

array — Definition

An **array** is a concatenation of data elements, all of which are of the same type. All the elements of an array are stored consecutively in memory, and each element within the array can be addressed by the array name plus a subscript.

For example, the array **int foo[3]** has three elements, each of which is an **int**. The three **ints** are stored consecutively in memory, and each can be addressed by the array name **foo** plus a subscript that indicates its place within the array, as follows: **foo[0]**, **foo[1]**, and **foo[2]**. Note that the numbering of elements within an array always begins with '0'.

Arrays, like other data elements, may be automatic (**auto**), **static**, or external (**extern**).

Arrays can be multi-dimensional; that is to say, each element in an array can itself be an array. To declare a multi-dimensional array, use more than one set of square brackets. For example, the multi-dimensional array **foo[3][10]** is a two-dimensional array that has three elements, each of which is an array of ten elements. The second sub-script is always necessary in a multi-dimensional array, whereas the first is not. For example, the form **foo[][10]** is acceptable, whereas **foo[10][]** is not. The first form is an indefinite number of ten-element arrays, which is correct C, whereas the second form is ten copies of an indefinite number of elements, which is illegal.

You can initialize automatic arrays and structures, provided that you know the size of the array, or of any array contained within a structure. An automatic array is initialized in the same manner as aggregate, but initialization is performed on entry to the routine at run time, instead of at compile time.

Flexible Arrays

A **flexible array** is one whose length is not declared explicitly. Each has exactly one empty '[ ]' array-bound declaration. If the array is multidimensional, the flexible dimension of the array must be the *first* array bound in the declaration; for example:

```
int example1[][20];   /* RIGHT */
int example2[20][];   /* WRONG */
```

The C language allows you to declare an indefinite number of array elements of a set length, but not a set number of array elements of an indefinite length.

Flexible arrays occur in only a few contexts; for example, as parameters:

```
char *argv[];
char p[][8];
```

as **extern** declarations:

```
extern int end[];
```

or as a member of a structure — usually, though not necessarily, the last:

```
struct nlist {
     struct nlist *next;
     char name[];
};
```

Example

The following program initializes an automatic array, and prints its contents.

```
main()
{
     int foo[3] = { 1, 2, 3 };

     printf("Here's foo's contents: %d %d %d\n",
           foo[0], foo[1], foo[2]);
}
```

See Also

definitions, initialization, struct
The C Programming Language, pages 25, 83, 210

as — Command

i80X86 assembler
as [-glx] [-o*file* **]***file* ...

as is the Mark Williams assembler. It is a multipass assembler that turns files of assembly language into relocatable object modules similar to those produced by the compiler. **as** is designed for writing small assembly-language subroutines. Because it is not intended to be used for full-scale assembly-language programming, it lacks many of the more elaborate facilities of full-fledged assemblers. For example, there are no facilities for conditional compilation or user-defined macros. However, it does optimize span-dependent instructions (for example, branches).

Features

as includes the following features:

- It automatically compiles jump instructions into either regular (three-byte) jumps or short (two-byte) jumps, whichever is required. There is no explicit short jump instruction.
- The assembler supports temporary labels, which conserves symbol table space and relieves you of having to invent many unique labels.
- Program modules are relocatable. They can be linked with each other and with C program modules produced by the COHERENT compiler. All assembled modules must be linked before they can be executed.
- The assembler does not support file inclusion, but multiple source files can be concatenated and assembled by including their names in the command line to run the assembler.
- The assembler generates SMALL model objects in the COHERENT **l.out** object format.

Usage

Normally, the assembler is invoked via the **cc** command, which will automatically assemble and link any file of source code that has the suffix **.s**. If you wish, however, you can invoke the assembler as a separate program, by using the following command line:

as [**-glx**] [**-o** *file*] *file* ...

The named *files* are concatenated and the resulting object code is written to the file specified by the **-o** option, or to file **l.out** if no **-o** option is given.

The option **-g** causes all symbols that are undefined at the end of the first pass to be given the type **undefined external**, as though they had been declared with a **.globl** directive.

The option **-l** tells the assembler to generate a listing. It writes the listing to the standard output, normally the terminal; it may be easily redirected to a file or printer using the **>** operator.

The option **-x** strips from the symbol table of the object module all non-global symbols that begin with the character 'L'. This speeds up the loading of files by removing compiler-generated labels from the symbol table.

Register Names

The following lists the identifiers that represent the i8086 machine registers, which are predefined:

```
AX   SP   AL   AH   CS
BX   BP   BL   BH   DS
CX   SI   CL   CH   ES
DX   DI   DL   DH   SS
```

Lexical Conventions

Assembler tokens consist of identifiers (also known as "symbols" or "names"), constants, and operators.

An identifier is a sequence of alphanumeric characters (including the period '.' and the underscore '_'). The first character must not be numeric. Only the first 16 characters of the name are significant; the assembler throws away the remainder. Upper case and lower case are different. The machine instructions, assembly directives, and built-in symbols that are used frequently are in lower case.

Numeric constants are defined by the assembler by using the same syntax as the C compiler: a sequence of digits that begins with a zero '0' is an octal constant; a sequence of digits with a leading '0x' is a hexadecimal constant ('A' through 'F' have the decimal values 10 through 15); and any strings of digits that do not begin with '0' are interpreted as decimal constants.

A character constant consists of an apostrophe followed by an ASCII character. The constant's value is the ASCII code for the character, right-justified in the machine word. For example, an instruction to move the letter 'A' to the register **al** could be expressed in either of two equivalent ways:

```
mov   al,$0x41
mov   al,$'A
```

The dollar sign indicates a n immediate operand.

A blank space can be represented either **0x20** (its ASCII value in hexadecimal), or as an apostrophe followed by a space ('), which on paper looks like just an apostrophe alone.

The following gives the multi-character escape sequences that can be used in a character constant to represent special characters:

\b	Backspace	(0010)
\f	Formfeed	(0014)
\n	Newline	(0012)
\r	Carriage return	(0015)
\t	Tab	(0011)
\v	Vertical tab	(0013)
nnn	Octal value	(0*nnn*)

A blank space can be represented either as 0x20 (its ASCII value in hexadecimal), or as an apostrophe followed by a space ('), which on the page would look like just an apostrophe.

Blanks and Tabs

Blanks and tab characters may be used freely between tokens, but not within identifiers. A blank or a tabulation character is required to separate adjacent tokens not otherwise separated, e.g., between an instruction opcode and its first operand.

Comments

Comments are introduced by a slash ('/') and continue until the end of the line. All characters in comments are ignored by the assembler.

Program Sections

The assembler permits you to divide programs into sections, each corresponding (roughly) to a functional area of the address space. Each program section has its own location counter during assembly. There are eight program sections, subdivided into three groups containing code, data and tables:

code:	**shri**	Shared instruction
	bssi	Uninitialized instruction
	prvi	Private instruction
data:	**prvd**	Private data
	shrd	Shared data
	bssd	Uninitialized data
	strn	Strings
tables:	**symt**	Symbol table

All Mark Williams assemblers use the same set of sections. This increases the portability of programs between operating systems. Not all the sections are distinct under COHERENT, however; the meanings of the sections under (including hints as to how the C compiler uses them) are as follows:

shri (shared instruction) is the same as **prvi** (private instruction); the adjective *shared* refers to the sharing of physical memory between two or more concurrent processes. **prvi** is used for all code generated by the C compiler.

Similarly, there is no distinction between **shrd** and **prvd**. The compiler uses the latter for all external and static data that are explicitly initialized in a C program.

Uninitialized sections are actually initialized to zeros. The reason is that the C compiler uses the **bssd** (uninitialized data) section for external or static data that are not explicitly initialized: the C language guarantees that these data are in fact initialized to zeros. The **bssi** (uninitialized instruction) section is not used by the compiler.

The **strn** (strings) section is actually a special part of the data section, used by the C compiler to store string constants. It is synonymous with **prvd** under COHERENT.

The **symt** (symbol table) section contains the symbol table used by the linker. Both the C compiler and the assembler generate symbol tables that go in this section.

In most cases, you need not worry about what all these program sections are, and can simply write code under the keyword **.prvi** or **.shri**, and write data under the keyword **.prvd** or **.shrd**. You are advised not to place items in the **symt** section, as this section is used for internal communication among the C compiler, the assembler, and the linker.

At the end of assembly, the sections of a program are concatenated so that in the assembly listing the program looks like a monolithic block of code and data. All **code** sections are combined into the i8086 **code** segment, and all **data** sections into the i8086 **data** segment. The symbol table is not linked when the program is executed, and so is not assigned to any i8086 segment.

The Current Location

The special symbol '.' (period) is a counter that represents the current location. The current location can be changed by an assignment; for example:

```
. = .+START
```

The assignment must not cause the value to decrease, and it must not change the program section, i.e., the right-hand operand must be defined in the same section as the current section.

Expressions

An expression is a sequence of symbols representing a value and a program section. Expressions are made up of identifiers, constants, operators, and brackets. All binary operators have equal precedence and are executed in a strict left-to-right order (unless

altered by brackets).

Notice that square brackets, '[' and ']', group expression elements, because parentheses are used for indexed register addressing.

Types

Every expression has a type determined by its operands. The simplest operands are symbols. The types of symbols are as follows:

Undefined — A symbol is defined if it is a constant or a label, or when assigned a defined value; otherwise, it is undefined. A symbol may become undefined if it is assigned the value of an undefined expression. It is an error to assemble an undefined expression in pass 2. Pass 1 allows assembly of undefined expressions, but phase errors may be produced if undefined expressions are used in certain contexts, such as in a **.blkw** or **.blkb**.

Absolute — An absolute symbol is one defined ultimately from a constant or from the difference of two relocatable values.

Register — These are the machine registers.

Relocatable — All other user symbols are relocatable symbols in some program section. Each program section is a different relocatable type.

Each keyword in the assembler has a secret type that identifies it internally; however, all of these secret types are converted to absolute in expressions. Thus, any keyword may be used in an expression to obtain the basic value of the keyword. This is useful when employing the keywords that define machine instructions. The basic value of a machine operation is usually the opcode with any operand-specific bits set to zero.

Notice that the type of an expression does not include such attributes as length (word or byte), so the assembler will not remember whether you defined a particular variable to be a word or a byte. Addresses and constants have different types, but the assembler does not treat a constant as an immediate value unless it is preceded by a dollar sign '$'. If you use a constant where an address is expected, **as** will treat the constant like an address (and vice versa). It is up to you to distinguish between variables and addresses or immediate values.

Operators

The following figure shows various characters interpreted as operators in expressions.

+	Addition
-	Subtraction
*	Multiplication
-	Unary negation
~	Unary complement
^	Type transfer
|	Segment construction

You can group expressions by means of square brackets ('[' and ']'); parentheses are reserved for use in address mode descriptions.

Type Propagation

When operands are combined in expressions, the resulting type is a function of both the operator and the types of the operands. The operators '*', '~', and unary '-' can only manipulate absolute operands and always yield an absolute result.

The operator '+' signifies the addition of two absolute operands to yield an absolute result, and the addition of an absolute to a relocatable operand to yield a result with the same type

as the relocatable operand.

The binary operator '-' allows two operands of the same type, including relocatable, to be subtracted to yield an absolute result. It also allows an absolute to be subtracted from a relocatable, to yield a result with the same type as the relocatable operand.

The binary operator " yields a result with the value of its left operand and the type of its right operand. It may be used to create expressions (usually intended to be used in an assignment statement) with any desired type.

Statements

A program consists of a sequence of statements separated by newlines or by semicolons. There are four kinds of statements: null statements, assignment statements, keyword statements, and machine instructions.

Labels

You can precede any statement by any number of labels. There are two kinds of labels: *name labels* and *temporary labels*.

A name label consists of an identifier followed by a colon (:). The program section and value of the label are set to that of the current location counter. It is an error for the value of a label to change during an assembly. This most often happens when an undefined symbol is used to control a location counter adjustment.

A temporary label consists of a digit (**0** to **9**) followed by a colon (:). Such a label defines temporary symbols of the form **xf** and **xb**, where **x** is the digit of the label. References of the form **xf** refer to the first temporary label **x:** forward from the reference; those of the form **xb** refer to the first temporary label **x:** backward from the reference. Such labels conserve symbol table space in the assembler.

Null Statements

A null statement is an empty line, or a line containing only labels or a comment. Null statements can occur anywhere. They are ignored by the assembler, except that any labels are given the current value of the location counter.

Assignment Statements

An assignment statement consists of an identifier followed by an equal sign '=' and an expression. The value and program section of the identifier are set to that of the expression. Any symbol defined by an assignment statement may be redefined, either by another assignment statement or by a label. An assignment statement is equivalent to the **equ** keyword statement found in many assemblers.

Assembler Directives

Assembler directives give instructions to the assembler. Each directive keyword begins with a period, and in general they are followed by operands.

The following directives change the current program section to the named section:

.bssd
.bssi
.prvd
.prvi
.shrd
.shri
.strn
.symt

The current location counter is set to the highest previous value of the location counter for the selected section.

The following describes the directives in detail.

.ascii *string*
The first non-white space character, typically a quotation mark, after the keyword is taken as a delimiter. **as** assembles successive characters from the string into successive bytes until it encounters the next instance of this delimiter. To include a quotation mark in a string, use some other character for the delimiter.

It is an error for a newline to be encountered before reaching the final delimiter. You can use a multi-character constant in the string to represent newlines and other special characters.

.blkb *expression*
Assemble a block of bytes that are filled with zeroes. The block is *expression* bytes long.

.blkw*expression*
Assemble a block of words that are filled with zeroes. The block is *expression* words long.

.byte *expression* [, *expression*]
The *expressions* in the list are truncated to byte size and assembled into successive bytes. Expressions in the list are separated by commas.

.even Force alignment by inserting a null byte of data, if necessary, to set the location counter to the next even location.

.odd Force alignment by inserting a null byte of data, if necessary, to set the location counter to the next odd location.

.globl *identifier* [, *identifier*]
The identifiers in the comma-separated list are marked as global. If they are defined in the current assembly, they may be referenced by other object modules; if they are undefined, they must be resolved by the linker before execution.

.page Force the printed listing of your assembly-language program to skip to the top of a new page by inserting a form-feed character into the file. The title is printed at the top of the page.

.title *string*
Print *string* at the top of every page in the listing. This directive also causes the listing to skip to a new page.

.word *expression* [, *expression*]
Truncate *expressions* to word length and assemble the resulting data into successive words. Expressions in the list are separated by commas.

Address Descriptors

The following syntax is used for general source and destination address descriptors. The symbol 'r' refers to a register and the symbol 'e' to an expression. Please refer to the following figure.

Syntax	*Addressing Mode*	*Example*	
r	Register	mov	ax, cx
e	Direct address	mov	ax, 0800
(r)	Indexing, no displacement	mov	ax, (bx)
e(r)	Indexing with displacement	mov	ax, 2(bx)
(r,r)	Double indexing, no displacement	mov	ax, (bx, si)
e(r,r)	Double indexing with displacement	mov	ax, 2(bx, si)
$e	Immediate	mov	ax, $0800

Note that the dollar sign is always used to indicate an immediate value, even if the expression is a constant.

A direct address is interpreted as either a direct address or a PC-relative displacement, depending on the requirements of the instruction.

If an address descriptor indicates an indexing mode and the base expression is of type absolute, the assembler uses the shortest displacement length (zero, one, or two bytes) that can hold the expression's value. Relocatable base expressions, whose values cannot be completely determined until the program is loaded, are always assigned two-byte displacements.

Any address descriptor may be modified by a segment escape prefix. A segment escape prefix consists of a segment register name followed by a colon ':'. The escape causes the assembler to produce a segment override prefix that uses the specified segment register as an operand. The assembler does not produce segment override prefixes unless explicitly required by an instruction.

8086 Instructions

The following machine instructions are defined. The examples illustrate the general syntax of the operands. Combinations that are syntactically valid may be forbidden for semantic reasons.

The examples use the following references:

a	General address
al	AL register
ax	AX register
cl	CL register
d	Direct address
dx	DX register
e	Expression
$e	Immediate expression
m	Memory address (not an immediate)
p	Port address

as treats as ordinary one-byte machine operations some operations that the Intel assembler ASM86 handles with special syntax; these include the *lock* and *repeat* prefixes. **as** makes no attempt to prevent the generation of incorrect sequences of these prefix bytes.

Although every machine operation has a type and value associated with it, in most cases the value was chosen to help **as** format the machine instructions.

For more information on these instructions, see the Intel *ASM86 Assembly Language Reference Manual*.

aaa ASCII adjust AL after addition

aad		ASCII adjust AX before division
aam		ASCII adjust AX after multiply
aas		ASCII adjust AL after subtraction
adcb	*r, a*	Add with carry, byte
adc	*r, a*	Add with carry, word
adcb	*a, r*	Add with carry, byte
adc	*a, r*	Add with carry, word
adcb	*a, $e*	Add with carry, byte
adc	*a, $e*	Add with carry, word
addb	*r, a*	Add, byte
add	*r, a*	Add, word
addb	*a, r*	Add, byte
add	*a, r*	Add, word
addb	*a, $e*	Add, byte
add	*a, $e*	Add, word
andb	*r, a*	Logical and, byte
and	*r, a*	Logical and, word
andb	*a, r*	Logical and, byte
and	*a, r*	Logical and, word
andb	*a, $e*	Logical and, byte
and	*a, $e*	Logical and, word
call	*d*	Near call, PC-relative
cbw		Convert byte into word
clc		Clear carry flag
cld		Clear direction flag
cli		Clear interrupt flag
cmc		Complement carry flag
cmpb	*r, a*	Compare two operands, byte
cmp	*r, a*	Compare two operands, word
cmpb	*a, r*	Compare two operands, byte
cmp	*a, r*	Compare two operands, word
cmpb	*a, $e*	Compare two operands, byte
cmp	*a, $e*	Compare two operands, word
cmps		Compare string operands, bytes
cmpsb		Compare string operands, bytes
cmpsw		Compare string operands, words
cwd		Convert word to double
daa		Decimal adjust AL after addition
das		Decimal adjust AL after subtraction
decb	*a*	Decrement by one, byte
dec	*a*	Decrement by one, word
divb	*m*	Unsigned divide, byte
div	*m*	Unsigned divide, word
esc	*a*	Escape **0xD8**
hlt		Halt
icall	*a*	Near call, absolute offset at EA word
idivb	*m*	Signed divide, byte
idiv	*m*	Signed divide, word
ijmp	*a*	Jump short, absolute offset at EA word
imulb	*m*	Signed multiply, byte
imul	*m*	Signed multiply, word
inb	*al, p*	Input, byte
in	*ax, p*	Input, word
inb	*al, dx*	Input, byte

in	*ax, dx*	Input, word
incb	*a*	Increment by one, byte
inc	*a*	Increment by one, word
int	*e*	Call to interrupt
into		Call to interrupt, overflow
iret		Interrupt return
ja	*d*	Jump short if greater
jae	*d*	Jump short if greater or equal
jb	*d*	Jump short if less
jbe	*d*	Jump short if less or equal
jc	*d*	Jump short if carry
jcxz	*d*	Jump short if CX equals zero
je	*d*	Jump short if equal to
jg	*d*	Jump short if greater
jge	*d*	Jump short if greater or equal
jl	*d*	Jump short if less
jle	*d*	Jump short if less or equal
jmp	*d*	Jump short, PC-relative word offset
jmpb	*d*	Jump short, PC-relative byte offset
jmpl	*d*	Jump long
jna	*d*	Jump short if not above
jnae	*d*	Jump short if not above or equal
jnb	*d*	Jump short if not below
jnbe	*d*	Jump short if not below or equal
jnc	*d*	Jump short if not carry
jne	*d*	Jump short if not equal
jng	*d*	Jump short if not greater
jnge	*d*	Jump short if not greater or equal
jnl	*d*	Jump short if not less
jnle	*d*	Jump short if not less or equal
jno	*d*	Jump short if not overflow
jnp	*d*	Jump short if not parity
jns	*d*	Jump short if not sign
jnz	*d*	Jump short if not zero
jo	*d*	Jump short if overflow
jp	*d*	Jump short if parity
jpe	*d*	Jump short if parity even
jpo	*d*	Jump short if parity odd
js	*d*	Jump short if sign
jz	*d*	Jump short if zero
lahf		Load flags into AH register
lds	*r, a*	Load double pointer into DS
lea	*r, a*	Load effective address offset
les	*r, a*	Load double pointer into ES
lock		Assert BUS LOCK signal
lodsb		Load byte into AL
lods		Load byte into AL
lodsw		Load byte into AL
loop	*d*	Loop; decrement CX, jump short if CX less than zero
loope	*d*	Loop; decrement CX, jump short if CZ not zero and equal
loopne	*d*	Loop; decrement CX, jump short if CX not zero and not equal

loopnz	*d*	Loop; decrement CX, jump short if CZ not zero and ZF equals zero
loopz	*d*	Loop; decrement CX, jump short if CX not zero and zero
movb	*r, a*	Move, byte
mov	*r, a*	Move, word
movb	*a, r*	Move, byte
mov	*a, r*	Move, word
movb	*a, $e*	Move, byte
mov	*a, $e*	Move, word
movb	*a, s*	Move, byte
mov	*a, s*	Move, word
movb	*s, a*	Move, byte
mov	*s, a*	Move, word
movsb		Move string byte-by-byte
movs		Move string word-by-word
movsw		Move string word-by-word
mulb	*m*	Multiply, byte
mul	*m*	Multiply, word
negb	*a*	Two's complement negation, byte
neg	*a*	Two's complement negation, word
nop		No operation
notb	*a*	One's complement negation, byte
not	*a*	One's complement negation, word
orb	*r, a*	Logical inclusive OR, byte
or	*r, a*	Logical inclusive OR, word
orb	*a, r*	Logical inclusive OR, byte
or	*a, r*	Logical inclusive OR, word
orb	*a, $e*	Logical inclusive OR, byte
or	*a, $e*	Logical inclusive OR, word
outb	*p, al*	Output to port, byte
out	*p, ax*	Output to port, word
outb	*dx, al*	Output to port, byte
out	*dx, ax*	Output to port, word
pop	*m*	Pop a word from the stack
pop	*s*	Pop a word from the stack
popf		Pop from stack into flags register
push	*m*	Push a word onto the stack
push	*s*	Push a word onto the stack
pushf		Push flags register onto the stack
rclb	*a, $1*	Rotate left $1 times, byte
rclb	*a, cl*	Rotate left CL times, byte
rcl	*a, $1*	Rotate left $1 times, word
rcl	*a, cl*	Rotate left CL times, word
rcrb	*a, $1*	Rotate right $1 times, byte
rcrb	*a, cl*	Rotate right CL times, byte
rcr	*a, $1*	Rotate right $1 times, word
rcr	*a, cl*	Rotate right CL times, word
rep		Repeat following string operation
repe		Find nonmatching bytes
repne		Repeat, not equal
repnz		Repeat, not equal
repz		Repeat, equal
ret		Return from procedure

rolb	*a, $1*	Rotate left, byte
rolb	*a, cl*	Rotate left, byte
rol	*a, $1*	Rotate left, word
rol	*a, cl*	Rotate left, word
rorb	*a, $1*	Rotate right, byte
rorb	*a, cl*	Rotate right, byte
ror	*a, $1*	Rotate right, word
ror	*a, cl*	Rotate right, word
sahf		Store AH into flags
salb	*a, $1*	Shift left, byte
salb	*a, cl*	Shift left, byte
sal	*a, $1*	Shift left, word
sal	*a, cl*	Shift left, word
sarb	*a, $1*	Shift right, byte
sarb	*a, cl*	Shift right, byte
sar	*a, $1*	Shift right, word
sar	*a, cl*	Shift right, word
sbbb	*r, a*	Integer subtract with borrow, byte
sbb	*r, a*	Integer subtract with borrow, word
sbbb	*a, r*	Integer subtract with borrow, byte
sbb	*a, r*	Integer subtract with borrow, word
sbbb	*a, $e*	Integer subtract with borrow, byte
sbb	*a, $e*	Integer subtract with borrow, word
scasb		Compare string data, byte
scas		Compare string data, word
shlb	*a, $1*	Shift left, byte
shlb	*a, cl*	Shift left, byte
shl	*a, $1*	Shift left, word
shl	*a, cl*	Shift left, word
shrb	*a, $1*	Shift right, byte
shrb	*a, cl*	Shift right, byte
shr	*a, $1*	Shift right, word
shr	*a, cl*	Shift right, word
stc		Set carry flag
std		Set direction flag
sti		Set interrupt enable flag
stosb		Store string data, byte
stos		Store string data, byte or word
stosw		Store string data, word
subb	*r, a*	Integer subtraction, byte
sub	*r, a*	Integer subtraction, word
subb	*a, r*	Integer subtraction, byte
sub	*a, r*	Integer subtraction, word
subb	*a, $e*	Integer subtraction, byte
sub	*a, $e*	Integer subtraction, word
testb	*r, a*	Logical compare, byte
test	*r, a*	Logical compare, word
testb	*a, r*	Logical compare, byte
test	*a, r*	Logical compare, word
testb	*a, $e*	Logical compare, byte
test	*a, $e*	Logical compare, word
wait		Wait until BUSY pin is inactive
xcall	*d, d*	Far call, immediate four-byte address
xchgb	*r, a*	Exchange memory, byte

xchg	*r, a*	Exchange memory, word
xlcall		Far call, address at EA double word
xljmp		Jump far, address at memory double word
xjmp	*d, d*	Jump far, immediate four-byte address
xlat		Table look-up translation
xorb	*r, a*	Logical exclusive OR, byte
xor	*r, a*	Logical exclusive OR, word
xorb	*a, r*	Logical exclusive OR, byte
xor	*a, r*	Logical exclusive OR, word
xorb	*a, $e*	Logical exclusive OR, byte
xor	*a, $e*	Logical exclusive OR, word
xret		Return, intersegment

80286 Instructions

The following instructions implement 80286-specific actions. Programs that use them cannot be run on 8086-based machines.

pusha		Push all general registers
popa		Pop all general registers
insb		Input byte from port DX to ES:(DI)
ins		Input word from port DX to ES:(DI)
outsb		Output byte from port DX from ES:(DI)
outs		Output word from port DX from ES:(DI)
enter	*$e, $e*	Make stack frame for procedure
leave		Tear down stack frame for procedure
bound	*r, e*	Check array index against bounds
sldt	*a*	Store Local Descriptor Table Register
str	*a*	Store Task Register
lldt	*a*	Load Local Descriptor Table Register
ltr	*a*	Load Task Register
verr	*a*	Verify a segment for reading
verw	*a*	Verify a segment for writing
sgdt	*m*	Store Global Descriptor Table register
sidt	*m*	Store Interrupt Descriptor Table register
lgdt	*m*	Load Global Descriptor Table register
lidt	*m*	Load Interrupt Descriptor Table register
smsw	*a*	Store Machine Status Word
lmsw	*a*	Load Machine Status Word
lar	*r,a*	Load access rights byte
lsl	*r,a*	Load segment limit
clts		Clear Task Switched Flag
arpl		Adjust RPL field of Selector
push	*$e*	Push sign extended byte

Also the *$1* forms become *$e* on **rol**, **rolb**, **ror**, **rorb**, **sal**, **salb**, **shrb**, **shr**, and **shrb**. This is because 8086 task of shifting and rotating by an immediate value could only take an immediate value of 1; however, on the 80286 the immediate value may be up to 31.

i8087 Op Codes

The assembler can also generate object files for use with the i8087 mathematics co-processor. The following listing presents the assembly language op codes for this feature. **st0** indicates floating point register 0 and **st1** indicates any floating point register but 0; **d** is the same as in the above listing.

d	Direct address
st0	Floating point register 0
st1	Any floating point register *except* 0

The following lists the i8087 instructions:

fabs		Absolute value
fadd	*st0, st1*	Add real
fadd	*st1, st0*	Add real
ffadd	*d*	Add real, float
fdadd	*d*	Add real, double
faddp		Add real and pop
faddp	*st, st0*	Add real and pop
fbld	*d*	Load packed decimal (BCD)
fbstp	*d*	Store packed decimal (BCD) and pop
fchs		Change sign
fclex		Clear exception
fnclex		Clear exception
fcom		Compare real
ffcom	*d*	Compare real, float
fdcom	*d*	Compare real, double
fcomp		Compare real and pop
fcomp	st1	Compare real and pop
ffcomp	*d*	Compare real and pop, float
fdcomp	*d*	Compare real and pop, double
fcompp		Compare real and pop twice
fdecstp		Decrement stack pointer
fdisi		Disable interrupts
fndisi		Disable interrupts, no operands
fdiv	*st0, st1*	Divide real
fdiv	*st1, st0*	Divide real
ffdiv	*d*	Divide real, float
fddiv	*d*	Divide real, double
fdivp		Divide real and pop
fdivp	st1	Divide real and pop
fdivr	*st0, st1*	Divide real reversed
fdivr	*st1, st0*	Divide real reversed
ffdivr	*d*	Divide real reversed, float
fddivr	*d*	Divide real reversed, double
fdivrp		Divide real reversed and pop
fdivrp	*st1*	Divide real reversed and pop
feni		Enable interrupts
fneni		Enable interrupts, no operands
ffree	*st1*	Free register
fiadd	*d*	Integer add
fladd	*d*	Integer add, long
ficom	*d*	Integer compare
flcom	*d*	Integer compare, long
ficomp	*d*	Integer compare and pop

flcomp	*d*	Integer compare and pop, long
fidiv	*d*	Integer divide
fldiv	*d*	Integer divide, long
fidivr	*d*	Integer divide reversed
fldivr	*d*	Integer divide, long reversed
fild	*d*	Integer load
flld	*d*	Integer load, long
fqld	*d*	Integer load, quad
fimul	*d*	Integer multiply
flmul	*d*	Integer multiply, long
fincstp		Increment stack pointer
finit		Initialize processor
fninit		Initialize processor
fist	*d*	Integer store
flst	*d*	Integer store, long
fistp	*d*	Integer store and pop
flstp	*d*	Integer store and pop, long
fqstp	*d*	Integer store and pop, quad
fisub	*d*	Integer subtract
flsub	*d*	Integer subtract, long
fisubr	*d*	Integer subtract reversed
flsubr	*d*	Integer subtract reversed, long
fld	*st1*	Load real
ffld	*d*	Load real, float
fdld	*d*	Load real, double
ftld	*d*	Load real, temp
fldcw	*d*	Load control word
fldenv	*d*	Load environment
fldlg2		Load log(10)2
fldln2		Load log(e)2
fldl2e		Load log(2)e
fldl2t		Load log(2)10
fldpi		Load pi
fldz		Load +0.0
fld1		Load +1.0
fmul		Multiply real
fmul	*st0, st1*	Multiply real
ffmul	*st1, st0*	Multiply real, float
fdmul	*d*	Multiply real, double
fmulp	*d*	Multiply real and pop
fnop	st1	No operation
fpatan		Partial arctangent
fprem		Partial remainder
fptan		Partial tangent
frndint		Round to integer
frstor	*d*	Restore saved state
fsave	*d*	Save state
fnsave	*d*	Save state
fscale		Scale
fsetpm		Set protected mode
fsqrt		Square root
fst	*st1*	Store real
ffst	*d*	Store real, float
fdst	*d*	Store real, double

fstcw	*d*	Store control word
fnstcw	*d*	Store control word
fstenv	*d*	Store environment
fnstenv	*d*	Store environment
fstp	*st1*	Store real and pop
ffstp	*d*	Store real and pop, float
fdstp	*d*	Store real and pop, double
ftstp	*d*	Store real and pop, temp
fstsw	*d*	Store status word
fnstsw	*d*	Store status word
fsub	*st0, st1*	Subtract real
fsub	*st1, st0*	Subtract real
ffsub	*d*	Subtract real, float
fdsub	*d*	Subtract real, double
fsubp		Subtract real and pop
fsubp	*st1*	Subtract real and pop
fsubr	*d*	Subtract real reversed
ffsubr	*d*	Subtract real reversed, float
fdsubr	*d*	Subtract real reversed, double
fsubrp		Subtract real reversed and pop
fsubrp	*st1*	Subtract real reversed and pop
ftst		Test stack top against +0.0
fwait		Wait while 8087 is busy
fxam		Examine stack top
fxch	*st1*	Exchange registers
fxch		Exchange registers
fxtract		Extract exponent and significance
fyl2x		Y*log(2)X
fyl2xp1		Y*log(2)(X+1)

C Compiler Conventions

as is often used to write small functions that perform tasks not easily or efficiently done in C. Such functions are intended to be called from a C program. As long as the assembly language source code follows compiler conventions, the assembler routine will be fully compatible with C functions. These conventions are (1) the names of external variables and (2) calling functions.

Naming Conventions

The C compiler appends an underline character '_' to the end of every external declared in a C source file. When referring to any external variable or function declared in a C source file, append an underscore to the name. In a similar manner, when defining a function or variable in an assembly language source file that is to be accessed from a C source file, append an underline character.

Function-Calling Conventions

Function-calling conventions deal with how arguments are passed to functions, how values are returned, and which registers are used for special purposes and must be protected.

Arguments

Function arguments are passed on the stack. They are pushed by the calling function, which also removes them when the called function returns. Looking at the declaration of the function, the order in which they are pushed onto the stack is from right to left; that is, the C compiler pushes the argument list in reverse order of declaration. The instruction **call** to jump to the function also pushes the return address, so that when the called routine gains control the first argument is found at offset 2 from the stack pointer.

Integer and pointer arguments are word size, and are simply pushed with a **push** instruction. Characters, although byte size, are not passed as bytes. The C language requires that **char** variables be promoted to the type **int** before being passed. The promotion is signed or unsigned, depending on the type of the **char** variable. **long**s are pushed one word at a time; the higher-address word is pushed first. This ensures that the words of the **long** are in the correct order on the stack, because the stack grows toward low-addressed memory.

Passing **float**s, **double**s, or structure arguments is more involved. C requires **float**s to be promoted to and passed as **double**s, so this conversion must be performed first. **double**s and structures are passed so that as they sit on the stack, all bytes are in the correct order; this is analogous to the passing of **long**s. This means, for example, that **double**s may be pushed with four **push word** instructions, beginning with the highest addressed word in the 64-bit double, and ending with the lowest addressed word.

If in doubt about how to apply any of this, try writing a simple C program that uses what you need, and compile it with the **-VASM** option to the **cc** command. This produces an assembly-language version of the C program, which can be studied to see exactly what the compiler does, and mimicked to good effect.

Return Values

Functions return values in various registers according to their type. **int**s and pointers are returned in the **ax** register. **char**s are returned by first promoting them to **int**s and returning the result in the **ax** register; effectively, this means that **char**s are returned in the **al** register. **long**s are returned in the **dx:ax** register pair, with the most significant word (also the high-address word) in the **dx** register, and the least significant word in the **ax** register.

floats, **double**s, and structures are returned in a more complex fashion. C requires **float**s be returned as **double**s, so they are converted. **double**s are returned in a special eight-byte array named **_fpac** (of course, in assembly language the name is **_fpac_**). This array is defined by the compiler. In the event that a function returns a structure, the contents of the structure are saved in memory, and the function returns a pointer to that structure in the **ax** register. The calling function then moves the bytes into the actual destination.

Again, if in doubt about how to do this in assembly language, try compiling a function with assembly language output to see how the compiler does it.

Important Registers

Every function must preserve the value of the **bp** register, which is the caller's stack frame pointer. Also, the compiler uses the **si** and **di** registers for register variables, so they must be preserved.

Example of an Assembly Language Program

The following assembly language file, **strchar.s** defines a function **strchar** that returns the number of occurrences of a character in a string.

FILE: strchar.s

```
/
/
/  Count and return the occurrences
/  of a character in a string.
/
/     int
/     strchar(s, c)
/     char *s;
/     int c;
/
/

   .globl     strchar_   / Make the name known externally.

   strchar_:
        push    si         / Standard C function
        push    di         / linkage. Save the
        push    bp         / si, di, and bp registers
        mov     bp, sp     / and set up new frame pointer.

        mov     si, 8(bp)  / String ptr -> si.
        mov     bx, 10(bp) / Char -> bx (actually bl).
        sub     ax, ax     / Clear ax (count register).
        sub     cx, cx     / Clear cx.

   0:   movb    cl, (si)   / Get character from string.
        jcxz    2f         / End of string?
        cmpb    bl, cl     / No. Do chars match?
        jnz     1f         / No.
        inc     ax         / Yes. Increment count.

   1:   inc     si         / Bump string pointer
        jmp     0b         / and loop again.

   2:   pop     bp         / Standard C return
        pop     di         / linkage. Restore
        pop     si         / saved registers and
        ret                / go home.
```

The following C program, **main.c** uses **strchar** The assembly language listing that follows, **main.s** was produced from **main.c** by the **-VASM** option in **cc**. The listing has been edited, and comments added, to illustrate what is happening.

FILE: main.c

```
main()
{
    int n;
    n = strchar("aardvark", 'a');
}
```

FILE: main.s

```
        .shri                   / ''code'' program section.

        .globl   main_

main_:

        .strn                   / ''string'' program section.

L2:     .byte    0x61           / This is the string
        .byte    0x61           / ''aardvark''
        .byte    0x72
        .byte    0x64
        .byte    0x76
        .byte    0x61
        .byte    0x72
        .byte    0x6B
        .byte    0x00

        .shri                   / Back to ''code''

        push  si                / Standard C function
        push  di                / linkage.  Save registers,
        push  bp                / set up new frame pointer (bp),
        mov   bp, sp            / and make room on stack
        sub   sp, $0x02         / for the auto int, ''n''

        mov   ax, $0x61         / Push the
        push  ax                / character 'a'.
        mov   ax, $L2           / Push the address
        push  ax                / of the string ''aardvark''
        call  strchar_          / Function call.
        add   sp, $0x04         / Remove args from stack.
        mov   -0x02(bp), ax     / Assign result to auto 'n'.

        mov   sp, bp            / Standard C return
        pop   bp                / linkage.  Adjust stack
        pop   di                / pointer, then restore
        pop   si                / registers and
        ret                     / go home.
```

Diagnostics

All errors detected by the assembler are reported on the screen as an error message that is tagged with a line number. If a symbol is associated with the error message (for example, if a symbol is undefined), then the symbol's name is also given. If more than one input file appears on the command line, error messages are tagged with the name of the source file.

If a listing is generated, errors are reported on the listing in the same format, with the error flags at the left margin. The total number of errors is displayed on the screen at the end of the assembly.

For a full listing of **as** error messages, see the tutorial for the C compiler, which appears earlier in this manual.

See Also

calling conventions, cc, commands

ASCII — Technical Information

ASCII is an acronym for the American Standard Code for Information Interchange. It is a table of seven-bit binary numbers that encode the letters of the alphabet, numerals, punctuation, and the most commonly used control sequences for printers and terminals. ASCII codes are used on all microcomputers sold in the United States.

The following table gives the ASCII characters in octal, decimal, and hexadecimal numbers, their definitions, and expands abbreviations where necessary.

000	0	0x00	NUL	**<ctrl-@>**	Null character
001	1	0x01	SOH	**<ctrl-A>**	Start of header
002	2	0x02	STX	**<ctrl-B>**	Start of text
003	3	0x03	ETX	**<ctrl-C>**	End of text
004	4	0x04	EOT	**<ctrl-D>**	End of transmission
005	5	0x05	ENQ	**<ctrl-E>**	Enquiry
006	6	0x06	ACK	**<ctrl-F>**	Positive acknowledgement
007	7	0x07	BEL	**<ctrl-G>**	Bell
010	8	0x08	BS	**<ctrl-H>**	Backspace
011	9	0x09	HT	**<ctrl-I>**	Horizontal tab
012	10	0x0A	LF	**<ctrl-J>**	Line feed
013	11	0x0B	VT	**<ctrl-K>**	Vertical tab
014	12	0x0C	FF	**<ctrl-L>**	Form feed
015	13	0x0D	CR	**<ctrl-M>**	Carriage return
016	14	0x0E	SO	**<ctrl-N>**	Shift out
017	15	0x0F	SI	**<ctrl-O>**	Shift in
020	16	0x10	DLE	**<ctrl-P>**	Data link escape
021	17	0x11	DC1	**<ctrl-Q>**	Device control 1 (XON)
022	18	0x12	DC2	**<ctrl-R>**	Device control 2 (tape on)
023	19	0x13	DC3	**<ctrl-S>**	Device control 3 (XOFF)
024	20	0x14	DC4	**<ctrl-T>**	Device control 4 (tape off)
025	21	0x15	NAK	**<ctrl-U>**	Negative acknowledgement
026	22	0x16	SYN	**<ctrl-V>**	Synchronize
027	23	0x17	ETB	**<ctrl-W>**	End of transmission block
030	24	0x18	CAN	**<ctrl-X>**	Cancel
031	25	0x19	EM	**<ctrl-Y>**	End of medium
032	26	0x1A	SUB	**<ctrl-Z>**	Substitute
033	27	0x1B	ESC	**<ctrl-[>**	Escape
034	28	0x1C	FS	**<ctrl-\>**	Form separator
035	29	0x1D	GS	**<ctrl-]>**	Group separator
036	30	0x1E	RS	**<ctrl-^>**	Record separator
037	31	0x1F	US	**<ctrl-_>**	Unit separator
040	32	0x20	SP	Space	
041	33	0x21	!	Exclamation point	
042	34	0x22	"	Quotation mark	
043	35	0x23	#	Pound sign (sharp)	
044	36	0x24	$	Dollar sign	
045	37	0x25	%	Percent sign	
046	38	0x26	&	Ampersand	
047	39	0x27	'	Apostrophe	
050	40	0x28	(	Left parenthesis	
051	41	0x29	)	Right parenthesis	
052	42	0x2A	*	Asterisk	

053	43	0x2B	+	Plus sign
054	44	0x2C	,	Comma
055	45	0x2D	-	Hyphen (minus sign)
056	46	0x2E	.	Period
057	47	0x2F	/	Virgule (slash)
060	48	0x30	0	
061	49	0x31	1	
062	50	0x32	2	
063	51	0x33	3	
064	52	0x34	4	
065	53	0x35	5	
066	54	0x36	6	
067	55	0x37	7	
070	56	0x38	8	
071	57	0x39	9	
072	58	0x3A	:	Colon
073	59	0x3B	;	Semicolon
074	60	0x3C	<	Less-than symbol (left angle bracket)
075	61	0x3D	=	Equal sign
076	62	0x3E	>	Greater-than symbol (right angle bracket)
077	63	0x3F	?	Question mark
0100	64	0x40	@	At sign
0101	65	0x41	A	
0102	66	0x42	B	
0103	67	0x43	C	
0104	68	0x44	D	
0105	69	0x45	E	
0106	70	0x46	F	
0107	71	0x47	G	
0110	72	0x48	H	
0111	73	0x49	I	
0112	74	0x4A	J	
0113	75	0x4B	K	
0114	76	0x4C	L	
0115	77	0x4D	M	
0116	78	0x4E	N	
0117	79	0x4F	O	
0120	80	0x50	P	
0121	81	0x51	Q	
0122	82	0x52	R	
0123	83	0x53	S	
0124	84	0x54	T	
0125	85	0x55	U	
0126	86	0x56	V	
0127	87	0x57	W	
0130	88	0x58	X	
0131	89	0x59	Y	
0132	90	0x5A	Z	
0133	91	0x5B	[	Left bracket (left square bracket)
0134	92	0x5C	\	Backslash
0135	93	0x5D	]	Right bracket (right square bracket)
0136	94	0x5E	^	Circumflex
0137	95	0x5F	_	Underscore
0140	96	0x60	`	Grave

0141	97	0x61	a	
0142	98	0x62	b	
0143	99	0x63	c	
0144	100	0x64	d	
0145	101	0x65	e	
0146	102	0x66	f	
0147	103	0x67	g	
0150	104	0x68	h	
0151	105	0x69	i	
0152	106	0x6A	j	
0153	107	0x6B	k	
0154	108	0x6C	l	
0155	109	0x6D	m	
0156	110	0x6E	n	
0157	111	0x6F	o	
0160	112	0x70	p	
0161	113	0x71	q	
0162	114	0x72	r	
0163	115	0x73	s	
0164	116	0x74	t	
0165	117	0x75	u	
0166	118	0x76	v	
0167	119	0x77	w	
0170	120	0x78	x	
0171	121	0x79	y	
0172	122	0x7A	z	
0173	123	0x7B	{	Left brace (left curly bracket)
0174	124	0x7C	|	Vertical bar
0175	125	0x7D	}	Right brace (right curly bracket)
0176	126	0x7E	~	Tilde
0177	127	0x7F	DEL	Delete

Files

/usr/pub/ascii

See Also

string, technical information

ascii.h — Header File

Define non-printable ASCII characters
#include <ascii.h>

ascii.h defines a set of manifest constants that describe the non-printable ASCII characters.

See Also

ASCII, header files

asctime() — Time Function (libc)

Convert time structure to ASCII string
#include <time.h>
#include <sys/types.h>
char *asctime(*tmp***) struct tm ****tmp***;**

asctime takes the data found in *tmp*, and turns it into an ASCII string. *tmp* is of the type **tm**, which is a structure defined in the header file **time.h**. This structure must first be initialized by either **gmtime** or **localtime** before it can be used by **asctime**. For a further

discussion of **tm**, see the entry for **time**.

asctime returns a pointer to where it writes the text string it creates.

Example

The following example demonstrates the functions **asctime**, **ctime**, **gmtime**, **localtime**, and **time**, and shows the effect of the environmental variable **TIMEZONE**. For a discussion of the variable **time_t**, see the entry for **time**.

```
#include <time.h>
#include <sys/types.h>
main()
{
     time_t timenumber;
     struct tm *timestruct;

     /* read system time, print using ctime */
     time(&timenumber);
     printf("%s", ctime(&timenumber));

     /* use gmtime to fill tm, print with asctime */
     timestruct = gmtime(&timenumber);
     printf("%s", asctime(timestruct));

     /* use localtime to fill tm, print with asctime */
     timestruct = localtime(&timenumber);
     printf("%s", asctime(timestruct));
}
```

See Also

time

Notes

asctime returns a pointer to a statically allocated data area that is overwritten by successive calls.

asin() — Mathematics Function (libm)

Calculate inverse sine

#include <math.h>
double asin(*arg***) double** *arg*;

asin calculates the inverse sin of *arg*, which must be in the range [-1., 1.]. The result will be in the range [-/2, /2].

Example

For an example of this function, see the entry for **acos**.

See Also

mathematics library

Diagnostics

Out-of-range arguments set **errno** to **EDOM** and return zero.

ASKCC — Environmental Variable

Force prompting for CC names
ASKCC=*YES/NO*

The environmental variable **ASKCC** directs the program **mail** to prompt for carbon-copy names. A carbon-copy (or CC) name gives another person to whom a mail message should be sent. To turn on prompting, use the command:

```
export ASKCC=YES
```

See Also

environmental variables, mail

assert() — Macro Diagnostics (assert.h)

Check assertion at run time
#include <assert.h>
void assert(*expression***) int** *expression***;**

assert checks the value of *expression*. If *expression* is false (zero), **assert** sends a message into the standard-error stream and calls **exit**. It is useful for verifying that a necessary condition is true.

The error message includes the text of the assertion that failed, the name of the source file, and the line within the source file that holds the expression in question. These last two elements consist, respectively, of the values of the preprocessor macros **__FILE__** and **__LINE__**.

assert calls **exit**, which never returns.

To turn off **assert**, define the macro **NDEBUG** prior to including the header **assert.h**. This forces **assert** to be redefined as

```
#define assert(ignore)
```

See Also

exit(), assert.h, C preprocessor

Notes

The Standard requires that **assert** be implemented as a macro, not a library function. If a program suppresses the macro definition in favor of a function call, its behavior is undefined.

Turning off **assert** with the macro **NDEBUG** will affect the behavior of a program if the expression being evaluated normally generates side effects.

assert is useful for debugging, and for testing boundary conditions for which more graceful error recovery has not yet been implemented.

assert.h — Header File

Define **assert()**
#include <assert.h>

assert.h is the header file that defines the macro **assert**.

See Also

assert(), header files

at — Device Driver

Drivers for hard-disk partitions

/dev/at* are the COHERENT system's AT devices for the hard-disk's partitions. Each device is assigned major-device number 11, and may be accessed as a block- or character-special device.

The **at** hard-disk driver handles two drives with up to four partitions each. Minor devices 0 through 3 identify the partitions on drive 0. Minor devices 4 through 7 identify the partitions on drive 1. Minor device 128 allows access to all of drive 0. Minor device 129 allows access to all of drive 1. To modify the offsets and sizes of the partitions, use the command **fdisk** on the special device for each drive (minor devices 128 and 129).

To access a disk partition through COHERENT, directory **/dev** must contain a device file that has the appropriate type, major and minor device numbers, and permissions. To create a special file for this device, invoke the command **mknod** as follows:

```
/etc/mknod /dev/at0a b 11   0 ; : drive 0, partition 0
/etc/mknod /dev/at0b b 11   1 ; : drive 0, partition 1
/etc/mknod /dev/at0c b 11   2 ; : drive 0, partition 2
/etc/mknod /dev/at0d b 11   3 ; : drive 0, partition 3
/etc/mknod /dev/at0x b 11 128 ; : drive 0, partition table
```

Drive Characteristics

When processing BIOS I/O requests prior to booting COHERENT, many IDE drives use "translation-mode" drive parameters: number of heads, cylinders, and sectors per track. These numbers are called translation-mode parameters because they do not reflect true physical drive geometry. The translation-mode parameters used by the BIOS code present on your host adapter can be obtained using the **dpb** utility found on the boot diskette of versions 3.2.0 and later of COHERENT. It is often necessary to patch the **at** driver with BIOS values of translation-mode parameters in order to boot COHERENT on IDE hard drives. In COHERENT versions 3.1.0 and later, drive parameters are stored in table **atparm_** in the driver. For the first hard drive, number of cylinders is a two-byte value at **atparm_+0**, number of heads is a single byte at **atparm_+2**, and number of sectors per track is a single byte at **atparm_+14**. For the second hard drive, number of cylinders is a two-byte value at **atparm_+16**, number of heads is a single byte at **atparm_+18**, and number of sectors per track is a single byte at **atparm_+30**. For example, if **testcoh** is a kernel linked with the **at** driver and you want to patch it for a second hard drive with 829 cylinders, 10 heads, and 26 sectors per track, you can do

```
/conf/patch testcoh atparm_+16=829 atparm_+18=10:c atparm_+30=26:c
```

To read the characteristics of a hard disk once the **at** driver is running, use the call to **ioctl** of the following form:

```
#include <sys/hdioctl.h>
hdparm_t hdparms;
      . . .
ioctl(fd, HDGETA, (char *)&hdparms);
```

where *fd* is a file descriptor for the hard disk device and *hdparms* receives the disk characteristics.

Non-Standard and Unsupported Types of Drives

Prior releases of the the COHERENT **at** hard-disk driver would not support disk drives whose geometry was not supported by the BIOS disk parameter tables. COHERENT adds support for these drives during installation by "patching" the disk parameters into the bootstrap and the **/coherent** image on the hard disk.

Files

/dev/at* — Block-special files
/dev/rat* — Character-special files

See Also

device drivers, fdisk

at — Command

Execute commands at given time
at [**-v**] [**-c** *command*] *time* [[*day*] *week*] [*file*]
at [**-v**] [**-c** *command*] *time month day* [*file*]

at executes commands at a given time in the future.

If the **-c** option is used, **at** executes the following *command*. If *file* is named, **at** reads the commands from it. If neither is given, **at** reads the standard input for commands.

If *time* is a one-digit or two-digit number, **at** interprets it as specifying an hour. If *time* is a three-digit or four-digit number, **at** interprets it as specifying an hour and minutes. If *time* is followed by **a**, **p**, **n**, or **m**, **at** assumes **AM**, **PM**, **noon**, or **midnight**, respectively; otherwise, it assumes that *time* indicates a 24-hour clock.

For example, the command

```
at -c "time | msg henry" 1450
```

set the **time** command to be executed at 2:50 PM, and pipe **time**'s output to the **msg** command, which will pass it to the terminal of user **henry**. Note that argument to the **-c** option had to be enclosed in quotation marks because it contains spaces and special characters; if this were not done, **at** would not be able to tell when the argument ended, and so would generate an error message. Also note that if you wish pass information to a user's terminal with the **at** command, you must tell **at** to whom to send the information. The command

```
at 250p commandfile
```

will set the file **commandfile** to be read and executed at 2:50 PM. Note that it is *not* necessary to use the file's full path name. Also, if the suffix **p** were not appended to the time, the file would be set to be read at 2:50 AM.

The time set in **at**'s command line is *not* the exact time that the command is executed. Rather, the daemon **cron** wakes up the file **/usr/lib/atrun** periodically to see if any commands have been scheduled commands to be executed at or before that time. The frequency with which **cron** executes **atrun** determines the "granularity" of **at** execution times; it may be changed by editing the file **/usr/lib/crontab**. For example, the entry

```
0,5,10,15,20,25,30,35,40,45,50,55 * * * *   /usr/lib/atrun
```

sets **/usr/lib/atrun** to be executed every five minutes. Thus, the **at** command that is set, for example, to 2:53 PM will actually be executed at 2:55 PM. **atrun** executes specified commands when it discovers that the given time is past; therefore, **at** commands are executed even if the system is down at the specified time or if the system's time is changed.

The **at** command has two forms, as shown above. In the first form, the option *day* names a day of the week (lower case, spelled out). If **week** is specified, **at** interprets the given *time* and *day* as meaning that time and day the following week. For example, the command

```
at -c "time | msg henry" 1450 friday week
```

executes **time** and sends its output to **henry**'s terminal one week from Friday at 2:50 PM.

In the second form given above, *month* specifies a month name (lower case, spelled out) and the number *day* specifies a day of the month. For example, the command

```
at 1450 july 4 commandfile
```

set the file **commandfile** to be read at 2:50 PM on July 4.

If the **-v** flag is given, **at** prints the time when the commands will be executed, giving you enough information to plan for the execution of the command. For example, if it is now August 13, 1990, at 2:30 PM, and you type the command

```
at -v -c "/usr/games/fortune | msg henry" 1435
```

at will reply:

```
Tue Aug 13 14:35:00
```

indicating that the command will be executed five minutes from now. However, if you type

```
at -v -c "/usr/games/fortune | msg henry" 1435 august 10
```

at will reply

```
Sun Aug 10 14:35:00 1991
```

which indicates that on Sunday, August 10 of next year, at 2:35 PM, the COHERENT system will print a **fortune** onto your terminal.

Should you create such a long-distance **at** file by accident, you can correct the error by simply deleting the file that encodes it from the directory **/usr/spool/at**. The file will be named after the time that it is set to execute, plus a unique two-character suffix, should more than one command be scheduled to run at the same time. For example, the file for the above command would be named **9108101435.aa**.

Finally, note that the current working directory, exported shell variables, file creation mask, user id, and group id are restored when the given command is executed.

Files

/bin/pwd — To find current directory
/usr/lib/atrun — Execute scheduled commands
/usr/spool/at — Scheduled activity directory
/usr/spool/at/ *yymmddhhmm.xx* — Commands scheduled at given time

See Also

at, commands, cron

atan() — Mathematics Function (libm)

Calculate inverse tangent
#include <math.h>
double atan(*arg***) double** *arg***;**

atan calculates the inverse tangent of *arg*, which may be any real number. The result will be in the range [-π/2, π/2].

Example

For an example of this function, see the entry for **acos**.

See Also

errno, mathematics library

atan2() — Mathematics Function (libm)

Calculate inverse tangent
double atan2(*num, den*) **double** *num, den*;

atan2 calculates the inverse tangent of the quotient of its arguments *num/den*. *num* and *den* may be any real numbers. The result will be in the range [-π, π]. The sign of the result will have the same sign as *num*, and the cosine will have the same sign as *den*.

Example

For an example of this function, see the entry for **acos**.

See Also

errno, mathematics library

ATclock — Command

Read or set the AT realtime clock
/etc/ATclock *[yy[mm[dd[hh[mm[.ss]]]]]]*

ATclock reads or sets the system realtime clock in an IBM PC-AT system. With no argument, it reads the realtime clock and returns a string in the format expected by the command **date**. With an argument, it sets the realtime clock to the given date.

The system startup files **/etc/brc** and **/etc/rc** typically contain a command of the form

```
date -s '/etc/ATclock'
```

to reset the time properly when the COHERENT system starts up. The AT realtime clock typically contains the current local standard time (not adjusted for daylight savings time).

See Also

commands, date, rc

atof() — General Function (libc)

Convert ASCII strings to floating point
double atof(*string*) **char** * *string*;

atof converts *string* into the binary representation of a double-precision floating point number. *string* must be the ASCII representation of a floating-point number. It can contain a leading sign, any number of decimal digits, and a decimal point. It can be terminated with an exponent, which consists of the letter **'e'** or **'E'** followed by an optional leading sign and any number of decimal digits. For example,

```
123e-2
```

is a string that can be converted by **atof**.

atof ignores leading blanks and tabs; it stops scanning when it encounters any unrecognized character.

Example

For an example of this function, see the entry for **acos**.

See Also

atoi(), atol(), float, general functions, long, printf(), scanf()

Notes

atof does not check to see if the value represented by *string* fits into a **double**. It returns zero if you hand it a string that it cannot interpret.

atoi() — General Function (libc)

Convert ASCII strings to integers
int atoi(*string***) char ***string***;**

atoi converts *string* into the binary representation of an integer. *string* may contain a leading sign and any number of decimal digits. **atoi** ignores leading blanks and tabs; it stops scanning when it encounters any non-numeral other than the leading sign, and returns the resulting **int**.

Example

The following demonstrates **atoi**. It takes a string typed at the terminal, turns it into an integer, then prints that integer on the screen. To exit, type **<ctrl-C>**.

```
main()
{
     extern char *gets();
     extern int atoi();
     char string[64];

     for(;;) {
          printf("Enter numeric string: ");
          if(gets(string))
               printf("%d\n", atoi(string));
          else
               break;
     }
}
```

See Also

atof(), atol(), general functions, int, printf(), scanf()

Notes

atoi does not check to see if the number represented by *string* fits into an **int**. It returns zero if you hand it a string that it cannot interpret.

atol() — General Function (libc)

Convert ASCII strings to long integers
long atol(*string***) char ***string***;**

atol converts the argument *string* to a binary representation of a **long**. *string* may contain a leading sign (but no trailing sign) and any number of decimal digits. **atol** ignores leading blanks and tabs; it stops scanning when it encounters any non-numeral other than the leading sign, and returns the resulting **long**.

Example

```
main()
{
     extern char *gets();
     extern long atol();
     char string[64];

     for(;;) {
          printf("Enter numeric string: ");
          if(gets(string))
               printf("%ld\n", atol(string));
          else
               break;
     }
}
```

See Also

atof(), atoi(), float, long, printf(), scanf()

Notes

No overflow checks are performed. **atol** returns zero if it receives a string it cannot interpret.

atrun — System Maintenance

Execute commands at a preset time

atrun is a program that executes programs at a time set by the command **at**.

When user **steve** types

```
at 1230 /v/steve/lunchtime
```

the command **at** creates a shell script in directory **/usr/spool/at** that contains the information needed to execute command **/v/steve/lunchtime** at a later time — in this instance, 12:30. The spooled file sits in **/usr/spool/at** until **/usr/lib/atrun** sees that the specified time has been reached, then it executes the spooled command and removes the entry from **/usr/spool/at**.

atrun is not a daemon; that is, it is invoked by another program, does its work and exits. Thus, it is typically run periodically from an entry in the file **/usr/lib/crontab**. See the article on **at** for more details.

See Also

at, system maintenance

Notes

Although **atrun** technically is a command, is never invoked directly by a user.

auto — C Keyword

Note an automatic variable

auto is an abbreviation for an *automatic variable*. This is a variable that applies only to the function that invokes it, and vanishes when the functions exits. The word **auto** is a C keyword, and must not be used to name any function, macro, or variable.

See Also

C keywords, extern, static, storage class

awk — Command

Pattern-scanning language
awk [-**y**][-**Fc**][-**f** *progfile*][*prog*][*file* ...]

awk is a general-purpose language designed for processing input data. Its features allow you to write programs that scan for patterns, produce reports, and filter relevant information from a mass of input data. It acts on each input *file* following the commands you write into an **awk** program. **awk** reads the standard input if no *file* is specified, which allows it to act as a filter in a pipeline; the *file* name '-' means the standard input.

You can specify the program either as an argument *prog* (usually enclosed in quotation marks to prevent interpretation by the shell **sh**) or in the form **-f** *progfile*, where *progfile* contains the **awk** program. If no **-f** option appears, the first non-option argument is the **awk** program *prog*.

The option flag **-y** specifies that any lower-case alphabetic character in a regular expression pattern should match both itself and the corresponding upper-case letter. This is identical to the matching found in the pattern-matching program **grep** with the **-y** option.

awk views its input as a sequence of records, each consisting of zero or more fields. By default, newlines separate records and white space (spaces or tabs) separates fields. The option **-Fc** changes the input field separator characters to the characters in the string *c*. An **awk** program can also change the field and record separators. The program can access the values of each field and the entire record through built-in variables.

For details on the construction of **awk** programs, consult the tutorial to **awk** that appears in this manual. Briefly, an **awk** program consists of one or more lines, each containing a *pattern* or an *action*, or both. A *pattern* determines whether **awk** performs the associated *action*. It may consist of regular expressions, line ranges, boolean combinations of variables, and beginning and end of input-text predicates. If no *pattern* is specified, **awk** executes the *action* (the pattern matches by default).

An *action* is enclosed in braces. The syntax of actions is C-like, and consists of simple and compound statements constructed from constants (numbers, strings), input fields, built-in and user-defined variables, and built-in functions. If an *action* is missing, **awk** prints the entire input record (line).

Unlike **lex** or **yacc**, **awk** does not compile programs into an executable image, but interprets them directly. Thus, **awk** is ideal for quickly-implemented, one-shot efforts.

Examples

The following examples illustrate the economy of expression of **awk** programs.

The first example prints all lines containing the string "COHERENT" (identical to **grep COHERENT**):

```
/COHERENT/
```

The built-in variable **NR** is the number of the current input record, so the following program prints the number of records (lines) in the input stream.

```
END   { print NR }
```

The built-in variable **$3** gives the value of the third field of the current record, so the following program sums the elements in column three of an input table and prints the total:

```
        { sum += $3 }
    END { print sum }
```

See Also

commands, grep, lex, sed, yacc
Introduction to the awk Language

Notes

There is no way to have a null field, such as is necessary to describe the colon-separated fields in **/etc/passwd.**

awk converts between strings and numbers automatically. Adding zero to a string forces **awk** to treat it as a number; concatenating "" to a number forces **awk** to treat it as a string.

Something like this seems to work

bad a /dev/at0b 18495 B

bad — Command

Maintain list of bad blocks
bad *option filesystem* [block ...]

A hard disk or floppy disk may have bad blocks on it: a "bad block" is a portion of disk that is flawed, and so cannot be used reliably because read or write errors occur on them. Bad blocks can result from microscopic flaws in the disk surface, and it is the unusual disk that is free of them. The COHERENT system keeps a list of bad blocks so it can avoid using them.

The command **bad** maintains the bad-block list for the given *filesystem*, which must be a block-special file. *option* must be exactly one of the characters **acdl**, which tell **bad** to do one of the following:

a Add each given *block* to the bad-block list
c Clear the bad-block list
d Delete each given *block* from the bad-block list
l List all blocks on the bad-block list

bad does not deallocate any i-node associated with a block when adding it to the bad-block list. You should run the command **icheck** with the **-s** option immediately after **bad** to correct the problem, or run the command **fsck**.

filesystem should be unmounted if possible. The user who invokes **bad** must have appropriate permissions for the given *filesystem*. For many file systems, only the superuser may use **bad** to change the bad-block list. Use the command **badscan** to create a prototype file.

When the **mkfs** command creates a file system, the prototype specification may include a bad block list for the new file system.

See Also

badscan, commands, icheck, mkfs, umount

badscan — Command

Build bad block list
/etc/badscan [**-v**] [**-o** *proto*] [**-b** *boot*] *device size*
/etc/badscan [**-v**] [**-o** *proto*] [**-b** *boot*] *device xdevice*

badscan scans a floppy disk or a partition of the hard disk for bad blocks. It writes onto the standard output a prototype file that lists all bad blocks on the disk.

badscan recognizes the following options:

-v Print an estimate of time needed to finish examining the device.

-o *proto* Redirect output into file *proto*.

-b *boot* Insert a given *boot* into the proto file as the bootstrap. The default is **/conf/boot**.

device names the special device to scan.

The command line for **badscan** comes in two forms, as shown at the top of this article. The first version is for a floppy disk; *size* gives the size of the device, in blocks. The second version is for a hard-disk partition; *xdevice* specifies devices **/dev/at0x** or **/dev/at1x**.

/etc/badscan -v -o /tmp/proto /dev/at0b /dev/at0x

LEXICON

15% done 6.6 minutes remaining

which uses the partition-table information in the master boot block of the drive to find the size of the **device**. Use **/dev/at0x** when examining a partition on hard-disk drive 0, and **/dev/at1x** when examining a partition on hard-disk drive 1.

See Also

at, commands, floppy disks, mkproto

Notes

The current implementation of **badscan** does not work with SCSI drives.

banner — Command

Print large letters
banner [*argument* ...]

banner prints large (seven-character by five-character) letters on the standard output. Each *argument* produces one large text output line. If there is no *argument*, each line from the standard input produces one line of large-text output.

See Also

commands, lpr, pr

basename — Command

Strip path information from a file name
basename *file* [*suffix*]

basename strips its argument *file* of any leading directory prefixes. If the result contains the optional *suffix*, **basename** also strips it. **basename** prints the result on the standard output.

For example, the command

```
basename /usr/fred/source.c
```

returns

```
source.c
```

basename is most useful when it is used with other shell commands. For example, the command

```
for i in *.c
do
     cp $i `basename $i .c`.backup
done
```

copies every file that has the suffix **.c** into an identically named file that has the suffix **.backup**.

See Also

commands, ksh, sh

bc — Command

Interactive calculator with arbitrary precision
bc [**-l**] [*file* ...]

bc is a language that performs calculations on numbers with an arbitrary number of digits. **bc** is most commonly used as an interactive calculator, where the user types arithmetic expressions in a syntax reminiscent of C. If **bc** is invoked with no *file* arguments on its

command line, it reads the standard input. For example:

```
Input              Output
bc
(1000+23)*42       42966
k = 2^10
16 * k             16384
2 ^ 100            1267650600228229401496703205376
```

bc may also be invoked with one or more *file* arguments. After **bc** reads each *file*, it reads the standard input. This provides a convenient way to access programs in files. A library of mathematical functions is available, obtained by using the **-l** option.

The following summarizes briefly the facilities provided by **bc**. More information is available in the tutorial to **bc** that is included with this manual.

Comments are enclosed between the delimiters '/*' and '*/'. Names of variables or functions must begin with a lower-case letter, and may have any number of subsequent letters or digits. Names may not begin with an upper-case letter because numbers with a base greater than ten may need need upper-case letters for their notation. The three built-in variables **obase**, **ibase**, and **scale** represent the number base for printing numbers (default, ten), the number base for reading numbers (default, ten), and the number of digits after the decimal (radix) point (default, zero), respectively. Variables may be simple variables or arrays, and need not be pre-declared, with the exception of variables internal to functions. Some examples of variables and array elements are **x25**, **array[10]**, and **number**.

Numbers are any string of digits, and may have one decimal point. Digits are taken from the ordinary digits (0-9) and then the upper-case letters (A-F), in that order.

Certain names are reserved for use as key words. The key words recognized by **bc** include the following:

if, for, do, while
: Test conditions and define loops, with syntax identical to C

break, continue
: Alter control flow within **for** and **while** loops.

quit
: Tell **bc** to exit immediately

define *function (arg, ..., arg)*
: Define a **bc** function by a compound statement, as in C.

auto *var, ..., var*
: Define variables that are local to a function, rather than having global scope.

return (*value*)
: Return a value from a function.

scale (*value*)
: Return the number of digits to the right of the decimal point in *value*.

sqrt (*value*)
: Return the square root of *value*

length (*value*)
: Return the number of decimal digits in *value*.

The following operators are recognized:

```
+     -     *     /     %     ^     ++
--    =     +=    -=    *=    /=    %=
^=    ==    !=    <     <=    >     >=
```

These operators are similar to those in C, with the exception of ^ and ^=, which are exponentiation operators. Expressions can be grouped with parentheses. Statements are separated with semicolons or newlines, and may be made into compound statements with braces. **bc** prints the value of any statement that is an expression but is not an assignment.

As in the editor **ed**, an '!' at the beginning of a line causes that line to be sent as a command to the COHERENT shell **sh**.

The built-in mathematics library contains the following functions and variables:

atan(*z*)	Arctangent of *z*
cos(*z*)	Cosine of *z*
exp(*z*)	Exponential function of *z*
j(*n*,*z*)	*n*th order Bessel function of *z*
ln(*z*)	Natural logarithm of *z*
pi	Value of pi to 100 digits
sin(*z*)	Sine of *z*

Examples

The first example calculates the factorial of its positive integer argument by recursion.

```
/*
 * Factorial function implemented by recursion.
 */
define fact(n) {
     if (n <= 1) return (n);
     return (n * fact(n-1));
}
```

The second example also calculates the factorial of its positive integer argument, this time by iteration.

```
/*
 * Factorial function implemented by iteration.
 */
define fact(n) {
     auto result;

     result = 1;
     for (i=1; i<=n; i++) result *= i;
     return (result);
}
```

Files

/usr/lib/lib.b — Source code for the library

See Also

commands, conv, dc, multi-precision arithmetic
bc Desk Calculator Language, tutorial

Notes

Line numbers do not accompany error messages in source files.

bind — Command

Bind key sequence to editing command
bind [-m] [*string* [= *command*]]

The command **bind** is used by the Korn shell **ksh** to bind one of its command-line editing commands to a given key sequence. The editing commands are used by **ksh** to perform its MicroEMACS-style command-line editing.

When **bind** is invoked without arguments, it lists on the standard output all current bindings.

When invoked with the syntax *string=command*, it binds the key-sequence *string* to the command. For example, the command

```
bind ^[X=end-of-line
```

binds the editing command **end-of-line** (which moves the cursor to the end of the command line) with the key sequence **<esc>X**. Note that **<esc>** is written **^[** — that is, a literal carat '^' followed by '['.

When invoked with the syntax **-m** *string=commands* , *string* is bound to *commands*, which contains one or more editing commands. This form of the **bind** command lets you build keyboard macros that combine several editing commands into one keystroke sequence.

For the list of editing commands, their default bindings, and other details of using MicroEMACS-style command-line editing, see the Lexicon entry for **ksh**.

See Also

commands, ksh

bit — Definition

bit is an abbreviation for "binary digit". It is the basic unit of data processing. A bit can have a value of either zero or one. Bits can be concatenated to form bytes.

A bit can be used either as a placeholder to construct a number with an absolute value, or as a flag whose value has a particular meaning under specially defined circumstances. In the former use, a string of bits builds an integer. In the latter use, a string of bits forms a **map**, in which each bit has a meaning other than its numeric value.

See Also

bit map, byte, definitions, nybble

bit-fields — Definition

A *bit-field* is a member of a structure or **union** that is defined to be a cluster of bits. It provides a way to represent data compactly. For example, in the following structure

```
struct example {
        int member1;
        long member2;
        unsigned int member3 :5;
}
```

member3 is declared to be a bit-field that consists of five bits. A colon ':' precedes the integral constant that indicates the *width*, or the number of bits in the bit-field. Also, the bit-field declarator must include a type, which must be one of **int**, **signed int**, or **unsigned int**.

A bit-field that is not given a name may not be accessed. Such an object is useful as "padding" within an object so that it conforms to a template designed elsewhere.

A bit-field that is unnamed and has a length of zero can be used to force adjacent bit-fields into separate objects. For example, in the following structure

```
struct example {
        int member1;
        int member2 :5;
        int :0;
        int member3 :5;
};
```

the zero-length bit-field forces **member2** and **member3** to be written into separate objects.

Finally, it is illegal to take the address of a bit-field.

See Also

bit, bit map, byte, definitions

Notes

Because bit-fields have many implementation-specific properties, they are not considered to be highly portable. Bit-fields use minimal amounts of storage, but the amount of computation needed to manipulate and access them may negate this benefit. Bit-fields must be kept in integral-sized objects because many machines cannot directly access a quantity of storage smaller than a "word" (a word is generally used to store an **int**).

bit map — Definition

A **bit map** is a string of bits in which each bit has a symbolic, rather than numeric, value.

See Also

bit, byte, definitions
The C Programming Language, page 136

Notes

C permits the manipulation of bits within a byte through the use of bit-field routines. These generate code rather than calls to routines. Bit fields are generally less efficient than masking because they always generate masking and shifting.

block — Technical Information

A *block* is a mass of data that is read at one time. Blocks are different lengths under different operating systems; COHERENT defines a block as being **BSIZE** bytes long.

Information is read in blocks from block-special devices, such as the hard disk or floppy disks. This is done to increase the speed with which data are read from these devices;

reading characters one at a time, such as is done with character-special devices such as terminals or modems, would be too slow.

See Also

technical information

boot — Driver

Boot block for hard-disk partition/nine-sector diskette

To be bootable, a COHERENT file system must contain a boot block that lives in the first block of the file system. In addition, all hard disks must contain the master boot block **mboot** or an equivalent.

mboot is the primary hard-disk boot, which reads the hard-disk partition table in sector 0 of the disk. If you type character '0' to '7' while booting your computer, **mboot** boots the designated hard-disk partition. If a partition is designated as active in the partition table, **mboot** boots it if you type no other character during booting. In either case, the boot block (first sector) of the selected partition must contain the appropriate bootstrap, such as "boot".

boot is a boot block for a hard-disk partition or a floppy disk. It must be installed as the first sector of the partition or diskette; for example, the following commands format and create a file system on a high-density, 5.25-inch diskette:

```
/etc/fdformat -v /dev/fha0
/etc/mkfs /dev/fha0 2400
/bin/cp /conf/boot.fha /dev/fha0
```

boot searches its root directory '/' for file **autoboot**. If it finds this kernel, **boot** loads and runs it. Otherwise, it gives the prompt ?, to which the user must type the name of the operating-system kernel to load (typically, "coherent"). If **boot** cannot find the requested kernel or if an error occurs, **boot** does not print an error message, but re-prompts with '?'.

Creating a Bootable Floppy Disk

In some cases, it is necessary to create a mini-version of COHERENT that can be booted from a floppy disk. For example, you may wish to test a device driver or a tricky program written in assembly language; by using a floppy-disk version of COHERENT, you can test your program yet protect your hard disk's file systems from damage should something go wrong.

The following describes how to create a floppy-disk version of COHERENT. If your drive A is a 5.25-inch, high-density device, type:

```
export DEV=/dev/fha0
export BS=30b
```

If, however, your drive A is a 3.5-inch, high-density device, type:

```
export DEV=/dev/fva0
export BS=36b
```

The rest of the commands will be the same for either device.

The next step is to copy disk 1 of your COHERENT release onto the hard disk. Insert disk 1 into drive A and type:

```
dd if=$DEV of=file_name bs=$BS count=80
```

where *file_name* is the file into which you are copying the disk.

Now, remove the COHERENT release disk and insert a blank floppy disk. The next step is to format it; type the following command:

```
/etc/fdformat $DEV
```

Now, copy the file that contains your copy of COHERENT release disk 1 onto the newly formatted floppy disk:

```
dd if=file_name of=$DEV bs=$BS count=80
```

where *file_name* is the file that holds COHERENT.

Now, mount the new floppy disk:

```
/etc/mount $DEV /f0
```

From this point on, you must have superuser status. If you have not yet done so, type

```
su root
```

and type the password if prompted.

The final steps further prepare the new floppy disk:

```
cd /f0
mv begin autoboot
rm Coh_3*
cd etc
rm brc*
rm build
```

The bootable floppy is now done. Type **<ctrl-D>** to return from superuser status, and then type the command

```
/etc/umount $DEV
```

to unmount the floppy disk. Put it away in a safe place.

Your bootable floppy disk contains about 700 blocks (350 kilobytes) of an information. We suggest that you put on it a handful of the most commonly used programs, such as the MicroEMACS screen editor.

Files

/conf/boot — Boot for AT partitions
/conf/boot.at — Boot for AT partitions (linked to **/conf/boot**)
/conf/boot.atx — AT master boot (linked to **/conf/mboot**)
/conf/boot.f9a — Boot for single-density, nine-sector, 5.25-inch floppy disk
/conf/boot.fha — Boot for 15-sector, 5.25-inch floppy disk
/conf/boot.fqa — Boot for quad-density, nine-sector, 3.5-inch floppy disk
/conf/boot.fva — Boot for 18-sector, 3.5-inch floppy disk
/conf/mboot — AT master boot

See Also

device drivers, fdisk, mboot, mkfs

boot.fha — Device Driver

Boot block for floppy disk

To be bootable, a COHERENT file system must contain a boot block (either **boot** or **boot.fha**). In addition, all hard disks must contain the master boot block **mboot** or an equivalent.

boot.fha is a boot block for a hard disk partition or a 15-sector floppy. It must be installed as the first sector of the partition or diskette, as follows:

```
/etc/fdformat -a /dev/fha0
/etc/badscan -v -o proto1 /dev/fha0 2400
/etc/mkfs /dev/fha0 proto1
rm proto1
cp /conf/boot.fha /dev/fha0
```

boot.fha searches its root directory '/' for file **autoboot**. If it finds this kernel, **boot.fha** loads and runs it. Otherwise, it gives the prompt ?, to which the user must type the name of the operating-system kernel to load (typically, **coherent**). If **boot.fha** cannot find the requested kernel or if an error occurs, **boot.fha** repeats the prompt and the user must type another name.

Files

/conf/boot.fha — Partition or 15-sector 96tpi floppy boot block

See Also

badscan, boot, device drivers, fdisk, mboot, mkfs

booting — Technical Information

How booting works

This article discusses the events that take place while starting, or *booting*, the COHERENT system. You do not need to read this article to know how to boot COHERENT, as all booting details are handled by COHERENT automatically. However, if you are interested in the details, or want to tailor COHERENT to your needs, it will help.

Two I/O devices are involved in bootstrapping. The first device is called the *boot* device; it contains the program necessary to invoke the COHERENT system and start it running. The second device is called the *root* device; it contains the root file system after the system is running. In most cases, these two devices are the same physical device.

Initial Startup

The initial installation of COHERENT loads information from floppy disk to the hard disk. The startup procedure first loads a small program from a floppy disk. This program, called the *bootstrap* program, then reads in the COHERENT system itself.

If the bootstrap finds a file called **autoboot** in the root directory of the device being booted, this program is loaded into memory and started. If this file does *not* exist, the bootstrap prompts you to type the name of the COHERENT image to boot, usually **coherent**. Note that **autoboot** is usually a link to file **coherent**.

After it loads the system image **/coherent** from the root device, the system initializes all devices, as well as starting the *idle* process and program **/etc/init**. The idle process uses any leftover computer time.

init controls the operation of the system from this point on. It first executes the command **/etc/brc** (i.e., boot **rc**), which may run commands like **fsck**. **brc** can request a reboot,

remain in single-user mode, or enter multi-user mode automatically. **init** then calls the *shell* to handle commands from the system console. The shell responds by prompting with **#**, expecting regular commands. At this time, the system is in *single-user mode*, meaning that no other users can log in to the system. The shell is running in superuser mode and only the console's user is logged in.

At this point, you can enter commands to the system in a normal fashion. One difference from normal, multi-user operation is that the system is in single-user mode, to allow special processing to take place before other users log in. Being in single-user mode gives you the opportunity to run **fsck** to check the file system and perform other administrative tasks before other users log into the system.

When administrative activities are finished, you should type **<ctrl-D>**. This terminates single user operation; **init** then opens the system to other users.

The file **/etc/rc** contains shell commands which the system executes just before making the system available to other users. This file typically includes commands to delete temporary files and mount standard devices. It also performs any installation-specific commands you require. As system administrator, you maintain this file. You must be sure that it is properly updated and never removed.

One command that must be included in **/etc/rc** is **/etc/update**, which periodically calls **sync()** to update buffered data to the disk.

init also maintains the file **/etc/utmp**, noting users' login and logout.

Files Used During Startup

The following files are used when the system is in single-user mode:

/etc/drvld	Load device drivers.
/etc/init	Initiate a process on each terminal line, call login when appropriate.
/etc/brc	Shell commands for booting.
/etc/checklist	List of partitions for **fsck**.
/bin/sh	Bourne shell.
/bin/ksh	Korne shell.

The following files are needed after the system has gone multi-user:

/etc/rc	Shell commands for multi-user startup.
/etc/ttys	Information about terminals.
/bin/login	Login program.
/etc/utmp	"who" file.
/etc/logmsg	Login prompt.
/etc/motd	Message of the day.
/etc/mount.all	Shell script to mount partitions.

See Also

technical information

boottime — System Maintenance

File that holds time system was last booted

/etc/boottime is an empty file maintained by the **init** process and the **date** command. The modification time of **boottime**, as displayed by the command **ls -l**, is the time that the system was last booted. You can read the time shown by **boottime** with **ls -l**, or with the system calls **stat** or **fstat**.

Files

/etc/boottime

See Also

date, init, mount, system maintenance

Notes

Commands that depend upon **/etc/boottime** may malfunction if the system's date is not set correctly. For instance, the **mount** command depends on the relative modification times of **/etc/boottime** and **/etc/mtab** to detect whether the mount table has been invalidated by a system boot. If the date is set sufficiently far into the past, the mount table may appear to be valid when in fact it is not.

brc — System Maintenance

Perform maintenance chores, single-user mode
/etc/brc

The shell script **/etc/brc** is executed by the **init** process when the COHERENT system enters single-user mode. The commands in **brc** do such things as set system clock, set the local time zone, and call **fsck** to scan and (if necessary) fix all file systems on your machine.

See Also

init, rc, system maintenance

break — Command

Exit from shell construct
break [*n*]

The command **break** is used with the shell to control how it performs loops. It is analogous to the **break** keyword in C.

When it is used without an argument, **break** forces the shell to exit from the innermost current **for**, **until**, or **while** loop. If used with an argument, **break** exits from *n* levels of **for**, **until**, or **while** loops.

The shell executes **break** directly.

See Also

commands, continue, for, ksh, sh, until, while

break — C Keyword

Exit from loop or switch statement

break is a C statement that causes an immediate exit from a **switch** sequence, or from a **while**, **for**, or **do** loop.

See Also

C keywords

brk() — COHERENT System Call (libc)

Change size of data area
brk(*addr*)
char **addr*;

The *break* is the lowest address above the data area of a process. **brk** sets the break to the given *addr*, possibly rounding up by some machine-dependent factor. It returns zero on success, -1 on failure.

See Also

COHERENT system calls, end, exec, malloc(), sbrk()

Diagnostics

brk sets **errno** to **ENOMEM** if the request fails.

bsearch() — General Function (libc)

Search an array
#include <stdlib.h>
char *bsearch(*key, array, number, size, comparison*)
char **key*, **array*;
size_t *number, size*;
int (**comparison*)();

bsearch searches a sorted array for a given item. *item* points to the object sought. *array* points to the base of the array; it has *number* elements, each of which is *size* bytes long. Its elements must be sorted into ascending order before it is searched by **bsearch**.

comparison points to the function that compares array elements. *comparison* must return zero if its arguments match, a number greater than zero if the element pointed to by *arg1* is greater than the element pointed to by *arg2*, and a number less than zero if the element pointed to by *arg1* is less than the element pointed to by *arg2*.

bsearch returns a pointer to the array element that matches *item*. If no element matches *item*, then **bsearch** returns NULL. If more than one element within *array* matches *item*, which element is matched is unspecified.

Example

This example uses **bsearch** to translate English into "bureaucrat-ese".

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct syntab {
     char *english, *bureaucratic;
} cdtab[] = {
/* The left column is in alphabetical order */
```

```
        "affect",       "impact",
        "after",        "subsequent to",
        "building",     "physical facility",
        "call",         "refer to as",
        "do",           "implement",

        "false",        "inoperative",
        "finish",       "finalize",
        "first",        "initial",
        "full",         "in-depth",
        "help",         "facilitate",

        "kill",         "terminate",
        "lie",          "inoperative statement",
        "order",        "prioritize",
        "talk",         "interpersonal communication",
        "then",         "at that point in time",
        "use",          "utilize"
};

int
comparator(key, item)
char *key;
struct syntab *item;
{
        return(strcmp(key, item->english));
}

main()
{
        struct syntab *ans;
        char buf[80];

        for(;;) {
                printf("Enter an English word: ");
                fflush(stdout);

                if(gets(buf) || !strcmp(buf, "quit") == NULL)
                        break;

                if((ans = bsearch(buf, (char *)cdtab,
                                sizeof(cdtab)/ sizeof(struct syntab),
                                sizeof(struct syntab),
                                comparator)) == NULL)
                        printf("%s not found\n");

                else
                        printf("Don't say \"%s\"; say \"%s\"!\n",
                                ans->english, ans->bureaucratic);
        }
}
```

See Also

general functions, qsort(), stdlib.h

Notes

The name **bsearch** implies that this function performs a binary search. A binary search looks at the midpoint of the array, and compares it with the element being sought. If that element matches, then the work is done. If it does not, then **bsearch** checks the midpoint of either the upper half of the array or of the lower half, depending upon whether the midpoint of the array is larger or smaller than the item being sought. **bsearch** bisects smaller and smaller regions of the array until it either finds a match or can bisect no further.

It is important that the input *array* be sorted, or **bsearch** will not function correctly.

buf.h — Header File

Buffer header
#include <sys/buf.h>

buf.h defines the structure used to hold buffers.

See Also

header files

buffer — Definition

A **buffer** is a portion of memory set aside to hold data read from or to be written to another process or device. Often, although not always, this involves setting aside a portion of the arena with **malloc** or its related functions.

Buffering, and problems therewith, are encountered most often when using the standard input and output (STDIO) routines. Many operating systems (including COHERENT) automatically place data from a peripheral device into a buffer. Buffers normally can be cleared with **fflush**, by pressing the carriage return key on routines that perform input, or by sending a newline character on routines that perform output. The function **fclose**, which closes a file stream, flushes all buffers associated with that stream. **exit** calls **fclose**.

Combining unbuffered and buffered I/O functions on the same file or device within one program will produce results that are at best unpredictable.

Example

The following example demonstrates what does and does not happen when you use **fflush** with the output buffer.

```
#include <stdio.h>
main()
{
	extern char *malloc();
	char *buffer;

	/* use malloc() to create a 120-char buffer */
	if ((buffer = malloc(120)) == NULL) {
		/* if malloc() fails, bail out */
		fprintf(stderr, "malloc failed\n");
		exit(1);
	}
```

```
        printf("Type your name:  ");
        fflush(stdout);
        gets(buffer);
        printf("Your name is %s\n", buffer);
}
```

See Also

arena, array, close, definitions, exit, fflush, malloc, STDIO

build — Command

Install COHERENT onto a hard disk
/etc/build

build installs COHERENT onto your hard disk. COHERENT runs **/etc/build** to install itself onto your hard disk. After installation, you should never have an occasion to run **build**.

See Also

commands

builtin — Command

Execute a command as a built-in command
builtin *command* [*arg* ...]

The command **ksh** is used by the Korn shell **ksh** to establish *command* as a built-in command.

See Also

commands, ksh

byte — Definition

A **byte** is a group of bits that encodes a character or a small-integer quantity. A byte, like a dollar, consists of eight bits.

The ANSI Standard defines the data type **char** as being equal to one byte. It defines all other data types as multiples of **char**.

See Also

bit, char, data formats, definitions, nybble

byte ordering — Technical information

Machine-dependent ordering of bytes

Byte ordering is the order in which a given machine stores successive bytes of a multibyte data item. Different machines order bytes differently.

The following example displays a few simple examples of byte ordering:

```
main()
{
     union
     {
          char b[4];
          int i[2];
          long l;
     } u;
     u.l = 0x12345678L;

     printf("%x %x %x %x\n",
          u.b[0], u.b[1], u.b[2], u.b[3]);
     printf("%x %x\n", u.i[0], u.i[1]);
     printf("%lx\n", u.l);
}
```

When run on "big-endian" machines, such as the M68000 or the Z8000, the program gives the following results:

```
12 34 56 78
1234 5678
12345678
```

As you can see, the order of bytes and words from low to high memory is the same as is represented on the screen.

However, when this program is run on "little-endian" machines, such as the PDP-11, you see these results:

```
34 12 78 56
1234 5678
12345678
```

As you can see, the PDP-11 inverts the order of bytes within words in memory.

Finally, when the program is run on the i8086 you see these results:

```
78 56 34 12
5678 1234
12345678
```

The i8086 inverts both words and long words.

See Also

C language, canon.h, data formats, technical information

C

c — Command

Print multi-column output
c [-l*N*] [-w*N*] [-012]

c reads lines from the standard input and writes them in columns on the standard output. The longest input line and the width of the page determine how many columns will fit across the page.

c recognizes the following options:

-l*N* Set the length of the page to *N* lines. **c** columnizes its output by pages when this option is used with mode 1 or mode 2.

-w*N* Set the width of the page to *N* characters. The default is 80.

-0 Multi-column mode 0. Order the fields horizontally across the page.

-1 Multi-column mode 1 (default mode). Order the fields vertically down each column; the last column may be short.

-2 Multi-column mode 2. Order the fields similarly to mode 1, but place blank fields in the last output line rather than the last column.

Options may also be given in the environmental variable **C**, separated by white space. Command line options override options in the environment. For example,

```
export C="-l56 -w72 -2"
c -w80 <file1
```

has the same effect as

```
c -l56 -w72 -2 -w80 <file1
```

This command sets the page width to 80 rather than to 72.

See Also

commands, export, pr

Diagnostics

c prints "out of memory" and returns an exit status of one if it cannot allocate enough memory to process its input.

C keywords — Overview

A **keyword** is a word that is reserved within C, and must not be used to name variables, functions, or macros. COHERENT recognizes the following C keywords:

alien	**extern**	**signed**
auto	**float**	**sizeof**
break	**for**	**static**
case	**goto**	**struct**
char	**if**	**switch**
const	**int**	**typedef**
continue	**long**	**union**
default	**readonly**	**unsigned**
do	**register**	**void**
double	**return**	**volatile**
else	**short**	**while**
enum		

In conformity with the ANSI standard, the keyword **entry** is no longer recognized. The keywords **const** and **volatile** are now recognized, but not implemented.

See Also

C language

C language — Overview

The following summarizes how COHERENT implements the C language.

Identifiers

Characters allowed: **A-Z**, **a-z**, **_**, **0-9**
Case sensitive.
Number of significant characters in a variable name:
at compile time: **128**
at link time: **16**
C appends '_' to end of external identifiers

Reserved Identifiers (Keywords)

See **C keywords**, above.

Data Formats (bits)

char	8
unsigned char	8
double	64
float	32
int	16
unsigned int	16
long	32
unsigned long	32
pointer	16
short	16
unsigned short	16

float *format:*

DECVAX floating point format:
1 sign bit
8-bit exponent
24-bit normalized fraction with hidden bit
DECVAX double format:
Same as **float**, but with 56 bits of fraction
Reserved values:
+- infinity, -0

All floating-point operations are done as **doubles**

Limits

Maximum bitfield size: 16 bits
Maximum number of **cases** in a **switch**: no formal limit
Maximum block nesting depth: no formal limit
Maximum parentheses nesting depth: no formal limit
Maximum structure size: 64 kilobytes
Maximum array size: 64 kilobytes

Preprocessor Instructions:

#define	**#ifdef**
#else	**#ifndef**
#elif	**#include**
#endif	**#line**
#if	**#undef**

Structure Name-Spaces

Supports both Berkeley and Kernighan-Ritchie conventions for structure in union.

Register Variables

Two available for **ints**
Two available for pointers

Function Linkage

Return values for **ints**: AX
Return values for **longs**: DX:AX
Return values for SMALL-model pointers: AX
Return values for LARGE-model pointers: DX:AX
Return values for **doubles** in DX:AX
Parameters pushed on stack in reverse order, **chars** and **shorts** pushed as words, **longs** and pointers pushed as **longs**, structures copied onto stack
Caller must clear parameters off stack
Stack frame linkage is done through SP register

Special Features and Optimizations

- Branch optimization is performed: this uses the smallest branch instruction for the required range.
- Unreached code is eliminated.
- Duplicate instruction sequences are removed.
- Jumps to jumps are eliminated.
- Multiplication and division by constant powers of two are changed to shifts when the results are the same.
- Sequences that can be resolved at compile time are identified and resolved.

See Also

argc, argv, C keywords, C preprocessor, header files, initialization, Lexicon, libraries, linker-defined symbols, main()

C preprocessor — Overview

Preprocessing encompasses all tasks that logically precede the translation of a program. The preprocessor processes headers, expands macros, and conditionally includes or excludes source code.

Directives

The C preprocessor recognizes the following directives:

#if	Include code if a condition is true
#elif	Include code if directive is true
#else	Include code if preceding directives fail
#endif	End of code to be included conditionally
#ifdef	Include code if a given macro is defined
#ifndef	Include code if a given macro is not defined
#define	Define a macro
#undef	Undefine a macro
#include	Read another file and include it
#line	Reset current line number

A preprocessing directive is always introduced by the '#' character. The '#' must be the first non-white space character on a line, but it may be preceded by white space and it may be separated from the directive name that follows it by one or more white space characters.

Preprocessing Operators

The Standard defines two operators that are recognized by the preprocessor: the "stringize" operator #, and the "token-paste" operator ##.

The operator # indicates that the following argument is to be replaced by a string literal; this literal names the preprocessing token that replaces the argument. For example, consider the macro:

```
#define display(x) show((long)(x), #x)
```

When the preprocessor reads the line

```
display(abs(-5));
```

it replaces it with the following:

```
show((long)(abs(-5)), "abs(-5)");
```

The ## operator performs "token pasting" — that is, it joins two tokens together, to create a single token. For example, consider the macro:

```
#define printvar(x) printf("%d\n", variable ## x)
```

When the preprocessor reads the line

```
printvar(3);
```

it translates it into:

```
printf("%d\n", variable3);
```

In the past, token pasting had been performed by inserting a comment between the tokens to be pasted. This no longer works.

Predefined Macros

The ANSI Standard describes the following macros that must be recognized by the preprocessor:

__DATE__	Date of translation
__FILE__	Source-file name
__LINE__	Current line within source file
__STDC__	Conforming translator and level
__TIME__	Time of translation

For more information on any one of these macros, see its entry.

Conditional Inclusion

The preprocessor will conditionally include lines of code within a program. The directives that include code conditionally are defined in such a way that you can construct a chain of inclusion directives to include exactly the material you want.

Macro Definition and Replacement

The preprocessor performs simple types of macro replacement. To define a macro, use the preprocessor directive **#define** *identifier value*. The preprocessor scans the translation unit for preprocessor tokens that match *identifier*; when one is found, the preprocessor substitutes *value* for it.

cpp

Under COHERENT, C preprocessing is done by the program **cpp**. The **cc** command runs **cpp** as the first step in compiling a C program. **cpp** can also be run by itself.

cpp reads each input *file*; it processes directives, and writes its product on **stdout**.

If the **-E** option is not used, **cpp** also writes into its output statements of the form **#line** *n filename*, so that the parser **cc0** can connect its error messages and debugger output with the original line numbers in your source files.

See the Lexicon entry on **cpp** for more information.

See Also

C language, cc, cpp

cabs() — Mathematics Function (libm)

Complex absolute value function
#include <math.h>
double cabs(*z*) **struct** { **double** *r*, *i*; } *z*;

cabs computes the absolute value, or modulus, of its complex argument *z*. The absolute value of a complex number is the length of the hypotenuse of a right triangle whose sides are given by the real part *r* and the imaginary part *i*. The result is the square root of the sum of the squares of the parts.

Example

For an example of this function, see the entry for **acos**.

See Also

hypot(), mathematics library

cal — Command

Print a calendar
cal [*month*] [*year*]

cal prints a calendar for the specified *year* (by default, the current year), or for the given *month* if one is specified. If neither is specified, a calendar of the current month is printed. *year* must be between 1 and 9999. *month* may be either the month name (lower case, spelled out or first three letters) or a number between 1 and 12.

For example, try:

```
cal september 1752
```

See Also

commands

Notes

cal assumes that the Gregorian calendar was adopted on September 3, 1752, which is the date of its adoption throughout the British empire.

calendar — Command

Reminder service
calendar [**-a**] [**-f***file*]... [**-d**[*date*]] [**-w**[*date*]] [**-m**[*month*]]

calendar is the COHERENT system's "reminder service". It reads a calendar file, which should contain information organized by date; if an event is scheduled to happen today or tomorrow, **calendar** prints the entry on the standard output. Thus, you can use **calendar** to remind you of both one-time events (such as appointments) and yearly events (such as anniversaries).

calendar recognizes the following command-line options:

-a	Search the calendars of all users and send mail. Default is to search only your calendar.
-f*file*	Search each "file" in order given. Default is **$HOME/.calendar**.
-d[*date*]	Print all entries for "date". Default date is today.
-w[*date*]	Print all entries for the week beginning with "date". Default is to print entries for today and tomorrow, with "tomorrow" encompassing the following Monday should "today" be a Friday.
-m[*month*]	Print entries for the given "month".

The following gives an example of a calendar file. Note that **calendar** understands different formats of dates:

```
Apr 16    Dave's birthday
7/6       Dad's birthday
Sep 26    Mom's birthday
Jun 30    Barry's birthday
10/4      Marianne's birthday
Jul 31    Anniversary!
Mar 16    Pot luck luncheon
```

Each user can run **calendar** by embedding the command

```
calendar
```

in his **.profile**.

If you wish, you can run **calendar** automatically for all users on your system, by inserting it into file **/usr/lib/crontab**. In this case, **calendar** should be used with its **-a** option, to force **calendar** to search every user's **$HOME** directory for a **.calendar** file and mail the appointments it finds to that user.

See Also

commands

Notes

calendar's notion of tomorrow understands weekends but not holidays. Thus, if you invoke **calendar** on a Friday, it returns the events for that day and the following Saturday, Sunday, and Monday. If Monday is a holiday, however, you will not receive appointments for Tuesday.

calling conventions — Technical Information

The following presents the calling conventions for COHERENT.

The design of the calling conventions had to take into account the fact that C does not require that the number of arguments passed to a function be the same as the number of arguments specified in the function's declaration. Routines with a variable number of arguments are not uncommon; for example, **printf** and **scanf** can take a variable number of arguments. Another consideration was the availability of **register** variables.

Therefore, COHERENT uses the following calling sequence. The function arguments are pushed onto the stack from the first, or rightmost, through the last, or leftmost. **longs** are pushed high-half first; this makes the word order compatible with the **dd** instruction. The function is then called with a near call. An **add** instruction after the call removes the arguments from the stack.

For example, the function call

```
int a;
long b;
char c;

foo()
{
        example(a, b, c);
}
```

generates the code

```
movb    al,c
cbw
push    ax
push    b+2
push    b
push    a
call    example_
add     sp,8
```

Note that an underbar character '_' has been appended to the function name. This serves two purposes. First, it makes it harder to accidentally call routines written in other languages. Second, it means that two routines with the same name can be called from C and another language in identical fashions.

The parameters and local variables in the called function are referenced as offsets from the **bp** register. The arguments begin at offset 8 and continue toward higher addresses, whereas the local variables begin at offset -2 and continue toward lower addresses.

The **sp** register points the local variable with the lowest address. Thus, when **example_** is reached in the above model, the stack frame resembles the following:

```
=========================   ← High
| c (widened to a word) |
|=======================|
|     high half of b    |
|=======================|
|      low half of b    |
|=======================|
|            a          |
=========================   ← Low
```

Functions return **int**s in the **ax** register, **long**s in the **dx:ax** register pair, pointers in the **ax** register and **double**s in **fpac_**.

The following program

```
example(a, b, c)
int a, b, c;
{
    return (a * b - c);
}
```

when compiled with the **-VASM** option, produces the following assembly-language code:

```
        .shri
        .globl example_

example_:
        push    si
        push    di
        push    bp
        mov     bp, sp
        mov     ax, 10(bp)
        imul    8(bp)
        sub     ax, 12(bp)
        pop     bp
        pop     di
        pop     si
        ret
```

The runtime startup initializes the registers **cs**, **ds**, **es**, and **ss**, and the segment registers remain unchanged. Other registers may be overwritten.

COHERENT pushes function arguments as follows.

char	Widened to **int**, then pushed
int	Pushed in machine word order
long	Pushed high order word, then low-order word
float	Widened to **double**, then pushed
double	Pushed high order, then low order
struct	Pushed in memory order
union	Pushed in memory order

Functions return values as follows:

char	In **al**
int	In **ax**
long	In **dx:ax**
float	Same as **double**
double	In **fpac_**
struct	Pointer in **ax**
union	Pointer in **ax**
pointer	In **ax**

A function that returns a **struct** or **union** actually returns a pointer; the code generated for the function call block-moves the result to its destination. Functions that return a **float** or **double** return it in the global double **fpac_**.

For example, consider the call

```
example(i, l, c, cp);
```

where **i** is an **int**, **l** is a **long**, **c** is a **char**, **cp** is a pointer to a **char**, and **example** declares two automatic **ints**. After execution of the call and the prologue of **example**, the stack contains the following 11 words:

```
=====================   ← High
|        cp         |
|===================|
|         c         |
|===================|
|  high word of l   |
|===================|
|   low word of l   |
|===================|
|         i         |
|===================|
|  return address   |
|===================|
|     saved SI      |
|===================|
|     saved DI      |
|===================|
|     saved BP      |
|===================|
| space for auto 1  |
|===================|
| space for auto 2  |
=====================   ← Low
```

The following example performs a simple function call:

```
main()
{
     example(1, 2);  /* call sample routine */
}

example(p1, p2)
{
     int a, b;

     a = 3;
     b = 4;
}
```

When the function **example** is about to return, the stack appears as follows:

```
=================          ← High
|       2       |          ← parm 2      10(bp)
|===============|
|       1       |          ← parm 1       8(bp)
|===============|
| Return Address |                        6(bp)
|===============|
|   main's SI   |                        4(bp)
|===============|
|   main's DI   |                        2(bp)
|===============|
|   main's BP   |                         (bp)
|===============|
|       3       |          ← a          -2(bp)
|===============|
|       4       |          ← SP b       -4(bp)
=================          ← Low
```

See Also

C language, technical information

calloc() — General Function (libc)

Allocate dynamic memory

char *calloc(*count, size***) unsigned** *count, size*;

calloc is one of a set of routines that helps manage a program's arena. **calloc** calls **malloc** to obtain a block large enough to contain *count* items of *size* bytes each; it then initializes the block to zeroes. When this memory is no longer needed, you can return it to the free pool by using the function **free**.

calloc returns the address of the chunk of memory it has allocated, or NULL if it could not allocate memory.

Example

This example attempts to **calloc** a small portion of memory; it then reallocates it to demonstrate **realloc**.

```
#include <stdio.h>

main()
{
     register char *ptr, *ptr2;
     extern char *calloc(), *realloc();
     unsigned count, size;

     count = 4;
     size = sizeof(char *);
```

```
        if ((ptr = calloc(count, size)) != NULL)
             printf("%u blocks of size %u calloced\n",
                     count, size);
        else
             printf("Insuff. memory for %u blocks of size %u\n",
                  count, size);

        if ((ptr2 = realloc(ptr,(count*size) + 1)) != NULL)
             printf("1 block of size %u realloced\n",
                  (count*size)+1);
}
```

See Also

arena, free(), general functions, malloc(), memok(), realloc(), setbuf()

candaddr() — General Function (libc)

Convert a daddr_t to canonical format

#include <canon.h>
#include <sys/types.h>
void candaddr(*s*)
daddr_t *s*;

candaddr performs canonical conversion upon a **daddr_t**. It returns nothing, and it is its own inverse. For details on canonical conversion, see **canon.h**.

Example

For an example of this function, see **canon.h**.

See Also

canon.h, general functions

candev() — General Function (libc)

Convert a dev_t to canonical format

#include <canon.h>
#include <sys/types.h>
void candev(*s*)
dev_t *s*;

candev performs canonical conversion upon a **dev_t**. It returns nothing, and it is its own inverse. For more information on canonical conversion, see **canon.h**.

See Also

canon.h, general functions

canino() — General Function (libc)

Convert an ino_t to canonical format

#include <canon.h>
#include <sys/types.h>
void canino(*s*)
ino_t *s*;

canino performs canonical conversion upon a **ino_t**. It returns nothing, and it is its own inverse. For more information on canonical conversion, see **canon.h**.

See Also

canon.h, general functions

canint() — General Function (libc)

Convert an int to canonical format
#include <canon.h>
#include <sys/types.h>
void canint(*s*)
int *s* ;

canint performs canonical conversion upon a **int**. It returns nothing, and it is its own inverse. For more information on canonical conversion, see **canon.h**.

See Also

canon.h, general functions

canlong() — General Function (libc)

Convert a long to canonical format
#include <canon.h>
#include <sys/types.h>
void canlong(*s*)
long *s* ;

canlong performs canonical conversion upon a **long**. It returns nothing, and it is its own inverse. For more information on canonical conversion, see **canon.h**.

See Also

canon.h, general functions

canon.h — Header file

Portable layout of binary data
#include <canon.h>
#include <sys/types.h>

The layout of binary data varies among machines. For example, the byte order of a 16-bit word on the PDP-11 is low-byte.high-byte, whereas on the Z8000 it is high-byte.low-byte.

To ensure that file systems can be ported among machines with differing byte orders, COHERENT uses a canonical layout of binary data. (The word "canonical" in this context means, "of or conforming to a general rule".) Data not in primary memory (e.g., on disk or communications line) must conform to COHERENT's canonical layout.

To insulate programs from the details of the difference between the 'natural' and canonical layouts, the COHERENT system provides a set of procedures to convert from one layout to another. They are as follows:

canshort() . . . . . Convert a **short**
canint(). . . . . Convert an **int**
canlong(). . . . . Convert a **long**
canvaddr() . . . . . Convert **vaddr_t**
cansize() . . . . . Convert **fsize_t**
candaddr(). . . . . Convert **daddr_t**
cantime(). . . . . Convert **time_t**
candev() . . . . . Convert **dev_t**
canino() . . . . . Convert **ino_t**

Each procedure takes an lvalue of the indicated type, converts it in place, and returns nothing. The argument should not have side-effects. Each procedure is its own inverse. Several procedures are designed for elements of file systems.

The file formats that contain canonical binary data and the commands that deal with them are as follows:

Format	*Commands*
ar.h	**ar, ld, ranlib**
dir.h	**ls, tar**
l.out.h	**as, cc, db, ld, nm, size, strip**

Any program that manipulates binary data within files must perform canonical conversion immediately upon input and immediately before output. The following fragment of the source code to the command **df** should be instructive:

```
#include <stdio.h>
#include <canon.h>
#include <filsys.h>
char    superb[BSIZE];

        .
        .
        .

df(fs)
char *fs;
{
        register struct filsys *sbp = &superb;
        FILE *fp;
        daddr_t nfree;

        if ((fp = fopen(fs, "r")) == NULL) {
                perror(fs);
                return (1);
        }

        fseek(fp, (long)BSIZE, 0);
        if (fread(superb, sizeof superb, 1, fp) != 1) {
                fprintf(stderr, "%s: read error\n", fs);
                return (1);
        }

        candaddr(sbp->s_tfree);
        candaddr(sbp->s_fsize);
        canshort(sbp->s_isize);
        nfree = sbp->s_tfree;

        if (nfree > sbp->s_fsize-sbp->s_isize  ||  nfree < 0) {
                fprintf(stderr, "%s: bad free count\n", fs);
                return (1);
        }
```

```
        printf("%s: %ld\n", fs, nfree);
        fclose(fp);
        return (0);
}
```

Files

<canon.h>

See Also

ar.h, byte ordering, candaddr(), candev(), canino(), canint(), canlong(), canshort(), cansize(), cantime(), canvaddr(), dir.h, l.out.h, header files

canshort() — General Function (libc)

Convert a short to canonical format

#include <canon.h>
#include <sys/types.h>
void canshort(*s*)
short *s*;

canshort performs canonical conversion upon a **short**. It returns nothing, and it is its own inverse. For more information on canonical conversion, see **canon.h**.

Example

For an example of this function, see **canon.h**.

See Also

canon.h, general functions

cansize() — General Function (libc)

Convert an fsize_t to canonical format

#include <canon.h>
#include <sys/types.h>
void cansize(*s*)
size_t *s*;

cansize performs canonical conversion upon a **size_t**. It returns nothing, and it is its own inverse. For more information on canonical conversion, see **canon.h**.

See Also

canon.h, general functions

cantime() — General Function (libc)

Convert a time_t to canonical format

#include <canon.h>
#include <sys/types.h>
void cantime(*s*)
time_t *s*;

cantime performs canonical conversion upon a **time_t**. It returns nothing, and it is its own inverse. For more information on canonical conversion, see **canon.h**.

See Also

canon.h, general functions

canvaddr() — General Function (libc)

Convert a vaddr_t to canonical format
#include <canon.h>
#include <sys/types.h>
void canvaddr(*s*)
vaddr_t *s*;

canvaddr performs canonical conversion upon a **vaddr_t**. It returns nothing, and it is its own inverse. For more information on canonical conversion, see **canon.h**.

See Also

canon.h, general functions

case — Command

Execute commands conditionally according to pattern
case *token* **in** [*pattern* [|*pattern*] ...) *sequence* ;;] ... **esac**

case is a construct that used by the shell. It tells the shell to execute commands conditionally, according to a pattern. It tests the given *token* successively against each *pattern*, in the order given. It then executes the commands in the *sequence* corresponding to the first matching pattern. Optional '|' clauses specify additional patterns corresponding to a single *sequence*. If no *pattern* matches the *token*, the **case** construct executes no commands.

Each *pattern* can include text characters (which match themselves), special characters '?' (which matches any character except newline) and '*' (which matches any sequence of non-newline characters), and character classes enclosed in brackets '[ ]'; ranges of characters within a class may be separated by '-'. In particular, the last *pattern* in a **case** construct is often '*', which will match any *token*.

The shell executes **case** directly.

Example

The following example prints a string in response to a command-line option:

```
case $1 in
     FOO)  echo "This is option FOO";;
     BAR)  echo "This is option BAR";;
     BAZ)  echo "This is option BAZ";;
     *)    echo "An asterisk marks the default option";;
esac
```

See Also

commands, ksh, sh

case — C Keyword

Introduce entry in switch statement

The C keyword **case** is a label within a **switch** statement. For example:

```
while ((int = getchar()) != EOF)
     switch (foo) {
          case 'q':
          case 'Q':
               exit(0);
          case ' ':
               n++;
          default:
               break;
     }
```

case labels each of the three possibilities recognized by the **switch** statement: a space, 'q', and 'Q'. The statements that follow a **case** statement behave as if they were enclosed within braces.

Note that a **case** statement is simply a label: it sets a point to which the **switch** statement jumps, and execution continues from that point. Once a **switch** statement jumps to the point marked by a given **case** label, execution continues until an **exit**, **break**, or **return** is read, or the closing brace of the **switch** statement is encountered.

See Also

break, C keywords, switch

cast — Definition

The **cast** operation "coerces" a variable from one data type to another.

There are two reasons to cast a variable. The first is to convert a variable's data into a form acceptable to a given function. For example, the function **hypot** takes two **doubles**. If the variables **leg_x** and **leg_y** are **floats**, the rules of C require that they be cast automatically to **double**. If the compiler did not do not do this, **hypot** would grab a **double**'s worth of memory: the four bytes of your **float**, plus four bytes of whatever happens to be sitting on the stack. The leads to results that are less than totally accurate.

The other reason to cast a variable is when you cast one type of pointer to another. For example,

```
char *foo;
int *bar;
bar = (int *)foo;
```

Although **foo** and **bar** are of the same length, you would cast **foo** in this instance to stop the C compiler from complaining about a type mismatch.

See Also

data formats, data types, definitions

cat — Command

Concatenate/print files
cat [**-u**][*file* ...]

cat copies each *file* arguments to the standard output. A '-' tells **cat** to read the standard input. If no *file* is specified, **cat** reads the standard input.

The **-u** option makes the output unbuffered. Otherwise, **cat** buffers the output in units of the machine's disk block size (e.g., 512 bytes).

See Also

commands

Notes

If you redirect **cat**'s the output to one of its input files, it will loop forever, reading from the file the text that it has just written into it: in effect, **cat** will chase its own tail endlessly.

caveat utilitor — Definition

Latin (sort of): "Let the user beware."

See Also

definitions

cc — Command

Compiler controller

cc [*compiler options*] *file* .... [*linker options*]

cc is the program that controls compilation of C programs. It guides files of source and object code through each phase of compilation and linking. **cc** has many options to assist in the compilation of C programs; in essence, however, all you need to do to produce an executable file from your C program is type **cc** followed by the name of the file or files that hold your program. It checks whether the file names you give it are reasonable, selects the right phase for each file, and performs other tasks that ease the compilation of your programs.

File Names

cc assumes that each *file* name that ends in **.c** or **.h** is a C program and passes it to the C compiler for compilation.

cc assumes that each *file* argument that ends in **.s** is in Mark Williams assembly language and processes it with the assembler **as**.

cc also passes all files with the suffixes **.o** or **.a** unchanged to the linker **ld**.

How cc Works

cc normally works as follows: First, it compiles or assembles the source files, naming the resulting object files by replacing the **.c** or **.s** suffixes with the suffix **.o**. Then, it links the object files with the C runtime startup routine and the standard C library, and leaves the result in file *file*. If only one object file is created during compilation, it is deleted after linking; however, if more than one object file is created, or if an object file of the same name existed before you began to compile, then the object file or files are not deleted.

Options

The following lists all of **cc**'s command-line options. **cc** passes some options through to the linker **ld** unchanged, and correctly interprets to it the options **-o** and **-u**.

A number of the options are esoteric and normally are not used when compiling a C program. The following are the most commonly used options:

-c	Compile only; do not link
-f	Include floating-point **printf**
-l*name*	Pass library **lib***name***.a** to linker
-o *name*	Call output file *name*
-V	Print verbose listing of **cc**'s action

-A MicroEMACS option. If an error occurs during compilation, **cc** automatically invokes the MicroEMACS screen editor. The error or errors are displayed in one window and the source code file in the other, with the cursor set to the line number indicated by the first error message. Typing **<ctrl-X>>** moves to the next error, **<ctrl-X><** moves to the previous error. To recompile, close the edited file with **<ctrl-Z>**. Compilation will continue either until the program compiles without error, or until you exit from the editor by typing **<ctrl-U>** followed by **<ctrl-X><ctrl-C>**.

-B[*string*]
Backup option. Use alternate versions of the compiler for **cc0**, **cc1**, **cc2**, and **cc3**. If *string* is supplied, **cc** prepends it to the names of the phases of the compiler to form the pathnames where these are found. Otherwise, **cc** prepends the name of the current directory. If a **-t** option was previously given, only the parts of the compiler specified by it are affected. Any number of **-B** and **-t** options may be used, with each **-t** option specifying the passes affected by the subsequent **-B** option. For example, the command

```
cc -tp2 -Bnew hello.c
```

compiles **hello.c** using **newcc2** in place of the ordinarily used **/lib/cc2**, and using **newcpp** in place of the ordinarily used **/lib/cpp**.

-c Compile option. Suppress linking and the removal of the object files.

-D*name*[=*value*]
Define *name* to the preprocessor, as if set by a **#define** directive. If *value* is present, it is used to initialize the definition.

-E Expand option. Run the C preprocessor **cpp** and write its output onto the standard output.

-f Floating point option. Include library routines that perform floating-point arithmetic. Because the floating-point routines require approximately five kilobytes of memory, the standard C library does not include them; the **-f** option tells the compiler to include them. If a program is compiled without the **-f** option but attempts to print a floating point number during execution by using the **e**, **f**, or **g** format specifications to **printf**, the message

```
You must compile with -f option for floating point
```

will be printed and the program will exit.

-I *name*
Include option. Specify a directory the preprocessor should search for files given in **#include** directives, using the following criteria: If the **#include** statement reads

```
#include "file.h"
```

cc searches for **file.h** first in the source directory, then in the directory named in the **-I***name* option, and finally in the system's default directories. If the **#include** statement reads

```
#include <file.h>
```

cc searches for **file.h** first in the directories named in the **-I***name* option, and then in the system's default directories. Multiple **-I***name* options are executed in the order of their of appearance.

-K Keep option. Do not erase the intermediate files generated during compilation. Temporary files will be written into the current directory.

-l *name*
library option. Pass the name of a library to the linker. **cc** expands **-l***name* into **/lib/lib***name***.a**. If an alternative library prefix has been specified by the **-tl** and **-B***string* options, then **-l***name* expands to *string***lib***name***.a**. Note that this is a *linker option*, and so must appear at the end of the **cc** command line, or it will not be processed correctly.

-M *string*
Machine option. Use an alternate version of **cc0**, **cc1**, **cc1a**, **cc1b**, **cc2**, **cc3**, **as**, **lib*.a**, and **crts0.o**, named by fixing *string* between the directory name and the pass and file names.

-n Instruct the linker **ld** to bind the output with separate shared and private segments, and which each starting on a separate hardware-segment boundary. This allows several processes to simultaneously use one copy of the shared segment. Note that programs linked with this option will run a little more slowly than if they were not so linked; however, if a program forks (e.g., **kermit**) or will be used by more than one user at a time (e.g., MicroEMACS), this slightly slower time will be more than offset by the program's being spared having to read an entire copy of itself from the disk.

-N[p0123sdlrt]*string*
Name option. Rename a specified pass to *string*. The letters **p0123sdlrt** refer, respectively, to **cpp**, **cc0**, **cc1**, **cc2**, **cc3**, the assembler, the linker, the libraries, the run-time start-up, and the temporary files.

-o *name*
Output option. Rename the executable file from the default to *name*. If this option is not used, the executable will be named after the first **.c** or **.o** file on the command line.

-O Optimize option. Run the code generated by the C compiler through the peephole optimizer. The optimizer pass is mandatory for the i8086, Z8000, and M68000 compilers, and need not be requested. It is optional for the PDP11 compiler, but is recommended for all files except those that consist entirely of initialized tables of data.

-q[p0123s]
Quit option. Terminate compilation after running the specified pass. The letters **p0123s** refer, respectively, to **cpp**, **cc0**, **cc1**, **cc2**, **cc3**, and the assembler. For example, to terminate compilation after running the parser **cc0**, type **-q0**.

-Q Quiet option. Suppress all messages.

-S Suppress the object-writing and link phases, and invoke the disassembler **cc3**. This option produces an assembly-language version of a C program for examination, for example if a compiler problem is suspected. The assembly-language output file name replaces the **.c** suffix with **.s**. This is equivalent to the **-VASM** option.

-t[p01ab23sdlrt]
Take option. Use alternate versions of the compiler phases and other files specified in the following string. If no following string is given, the **cc** uses alternate version of every phase of the compiler, except the preprocessor. If the **-t** option is followed by a **-B** option, **cc** prepends the prefix string named in the **-B** option to the phases and files named in the **-t** option; otherwise, the it looks for the alternate forms in the current directory.

-U *name*
Undefine symbol *name*. Use this option to undefine symbols that the preprocessor defines implicitly, such as the name of the native system or machine.

-V Verbose option. **cc** prints onto the standard output a step-by-step description of each action it takes.

V*string*
Variant option. Toggle (i.e., turn on or off) the variant *string* during the compilation. Variants that are marked **on** are turned on by default. Options marked **Strict:** generate messages that warn of the conditions in question. **cc** recognizes the following variants:

-VASM
Output assembly-language code. Identical to **-S** option, above. Default is **off**.

-VCOMM
Permit **.com**-style data items. Default is **on**.

-VFLOAT
Include floating point **printf** routines. Same as **-f** option, above.

-VPROF
Generate code to profile functions calls. Programs compiled with this option can be run with the **prof** command to print a summary of how much time the program spends in each subroutine, to help you optimize your programs. You must use this option to profile compile each module whose functions you wish to examine; and you must also use this option on the **cc** command line with which you link the program, to ensure that the appropriate library routines are linked into your executable.

-VQUIET
Suppress all messages. Identical to **-Q** option. Default is **off**.

-VSBOOK
Strict: note deviations from *The C Programming Language*, ed. 1. Default is **off**.

-VSCCON
Strict: note constant conditional. Default is **off**.

-VSINU
Implement struct-in-union rules instead of Berkeley-member resolution rules. Default is **off**, i.e., Berkeley rules are the default.

-VSLCON
Strict: **int** constant promoted to **long** because value is too big. Default is **on**.

-VSMEMB
Strict: check use of structure/union members for adherence to standard rules of C. Default is **on**.

-VSNREG
Strict: register declaration reduced to auto. Default is **on**.

-VSPVAL
Strict: pointer value truncated. Default is **off**.

-VSRTVC
Strict: risky types in truth contexts. Default is **off**.

-VSTAT
Give statistics on optimization.

-VS
Turn on all strict checking. Default is **on**.

-VSUREG
Strict: note unused registers. Default is **off**.

-VSUVAR
Strict: note unused variables. Default is **on**.

-V3GRAPH
Translate ANSI trigraphs. Default is **off**.

Linking Objects

The linker **ld** does not know about paths: it links exactly what you tell it to link via the **cc** command line. **cc** looks for compiler phases and for runtime startoff and library by searching the directories named in the environmental variable **LIBPATH**. If you do not define **LIBPATH** in your environment, it searches the default **LIBPATH** as defined in **/usr/include/path.h**. If you define **LIBPATH**, **cc** searches the directories in the order you specify. For example, a typical definition is:

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

This searches the current directory '.', then **/lib**, then **/usr/lib**.

See Also

as, C language, cc0, cc1, cc2, cc3, commands, cpp, ld
The C Language, tutorial

cc0 — Definition

cc0 is the COHERENT *parser*. It parses C programs using the method of recursive descent and translates the program into a logical tree format.

See Also

cc, cc1, cc2, cc3, cpp, definitions

cc1 — Definition

cc1 is the COHERENT code generator. This phase generates code from the trees created by the parser, **cc0**. The code generation is table driven, with entries for each operator and addressing mode.

See Also

cc, cc0, cc2, cc3, cpp, definitions

cc2 — Definition

cc2 is the optimizer/object generator phase of COHERENT. It optimizes the code generated by **cc1**, and writes the object code. COHERENT uses multiple optimization algorithms. One optimizes jump sequences: it eliminates common code, optimizes span-dependent jumps, and removes jumps to jumps. The other function scans the generated code repeatedly to eliminate unnecessary instructions.

See Also

cc, cc0, cc1, cc3, cpp, definitions

cc3 — Definition

cc3 is the output phase of COHERENT. It writes a file of assembly language rather than a relocatable object module. This phase is optional; it allows you to examine the code generated by the compiler. To produce an assembly-language output of a C program, use the **-S** option on the **cc** command line. For example,

```
cc -S foo.c
```

tells **cc** to produce a file of assembly language called **foo.s**, instead of an object module.

See Also

cc, cc0, cc1, cc2, cpp, definitions

cd — Command

Change directory
cd *directory*

The shell keeps track of the directory in which the user is currently working. If a command is not specified by a complete path name beginning with '/', the shell prefixes it with the name of the current working directory. **cd** changes the current working directory to *directory*. If no *directory* is specified, the directory named in the **$HOME** environmental variable becomes the current working directory.

See Also

commands, ksh, pwd, sh

ceil() — Mathematics Function (libm)

Set numeric ceiling
#include <math.h>
double ceil(*z*) **double** *z*;

ceil returns a double-precision floating point number whose value is the smallest integer greater than or equal to *z*.

Example

The following example demonstrates how to use **ceil**:

```
#include <math.h>
#include <stdio.h>
#define display(x) dodisplay((double)(x), #x)

dodisplay(value, name)
double value; char *name;
{
	if (errno)
		perror(name);
	else
		printf("%10g %s\n", value, name);
	errno = 0;
}
```

```
main()
{
     extern char *gets();
     double x;
     char string[64];

     for (;;) {
          printf("Enter number: ");
          if (gets(string) == NULL)
               break;
          x = atof(string);

          display(x);
          display(ceil(x));
          display(floor(x));
          display(fabs(x));
          display(sqrt(x));
     }
     putchar('\n');
}
```

See Also

abs(), fabs(), floor(), frexp(), mathematics library

cgrep — Command

Pattern search for C source programs
cgrep [-clnsA] [-r *new*] *expression file* ...

cgrep is a string-search utility. It resembles its cousins **grep** and **egrep**, except that it is specially designed to be used with C source files. It checks all C identifiers against *expression* and prints all lines in which it finds a match. **cgrep** allows you to search for a variable named 'i' without finding every 'if' and 'int' in your program. **cgrep** defines an "identifier" to be any variable name or C keyword. *expression* can be a regular expression; if it includes wildcard characters or '|'s, you must "quote it" to protect it against being modified by the shell. For details on the expressions that **cgrep** can recognize, see the Lexicon entry for **egrep**.

cgrep tests names that include the '.' and '->' operators against *expression*. Thus, to look for **ptr->val**, type:

```
cgrep "ptr->val" x.c
```

This finds **ptr->val** even if it contains spaces, comments, or is spread across lines. If it is spread across lines, it will be reported on the line that contains the last token. The only exception is if you include the **-A** option, in which case it will be reported on the line which contains the first token. This is to simplify MicroEMACS macros, as will be described below.

To find **structure.member**, type:

```
cgrep "structure\.member"
```

because '.' in a regular expression matches any character.

Do not include spaces in any pattern. Only identifiers and '.' or '->' between identifiers are included in the tokens checked for pattern-matching.

Command-line Options

cgrep recognizes the following command-line options:

-c Print all comments in each *file*. This form takes no expression.

-l List only the names of the files in which *expression* is found.

-n Prefix each line in which *expression* is found with its line number in the file.

-r Replace all expression matches with *new*. This option may not be used with any others, and it can only match simple tokens, not items like **ptr->val**. When **-r** is used and the input is **stdin**, a new file will always be created as **stdout**.

-s Print all strings in each *file*. This form takes no expression.

-A Write all lines in which *expression* is found into a temporary file. Then, call MicroEMACS with its error option to process the source file, with the contents of the temporary file serving as an "error" list. This option resembles the **-A** option to the **cc** command, and lets you build a MicroEMACS script to make systematic changes to the source file. To exit MicroEMACS and prevent **cgrep** from searching further, **<ctrl-U> <ctrl-X> <ctrl-C>**.

Examples

The command

```
cgrep tmp *.c
```

will find the variable name **tmp**, but not **tmpname**, or any occurrence of **tmp** in a string or comment.

The script

```
cgrep -c < myfile.c | wc -l
```

count the lines of comments in **myfile.c**.

The command

```
cgrep "x|abc|d" *.c
```

will find **x**, **ab**, or **d**. Note this is a regular expressions with a surrounding "^( )$" which is applied to every identifier. Thus, **reg*** will not match **register**, but **reg.*** will.

See Also

commands, egrep, grep, me

char — C Keyword

Data type

char is a C data type. It is the smallest addressable unit of data. According to the ANSI Standard, a **char** consists of exactly one byte of storage; a byte, in turn, must be composed of at least eight bits. **sizeof(char)** returns one by definition, with all other data types defined as multiples thereof. All Mark Williams compilers sign-extend **char** when it is cast to a larger data type.

Under COHERENT, a **char** by default is signed.

See Also

byte, C keywords, data formats, unsigned

chars.h — Header File

Character definitions
#include <sys/chars.h>

chars.h defines manifest constants for some commonly used characters.

See Also

header files

chase — Command

Highly amusing video game
/usr/games/chase [**-c**] [*speed*]

chase is a COHERENT version of a popular video game. It runs on the console of an IBM AT COHERENT system with input from the console keyboard. **chase** assumes that the system console is a monochrome display adapter unless you select the **-c** color-display option.

To accomodate different computer system speeds and different levels of skill, **chase** prompts the user to type a speed when the game begins. Press **<return>** to try out the game with the default speed of ten; typing a higher number makes the game slower, a lower number makes it faster. If you can play at speed zero on a fast computer system, you play too many video games. If you know the speed you want, you can enter it as a command-line argument. If you see the boss coming, quit by pressing **<ctrl-C>**.

The Rules

The player (represented by a blinking shaded rectangle) attempts to evade four "ghosts" (represented by shaded rectangles with arrows) while erasing dots from the playing-board maze.

At the beginning of a game, the four ghosts are in the *ghost box* above the center of the maze and the player is below it. The maze is filled with dots, including four blinking diamonds called *power pellets*. The ghosts emerge from the ghost box and chase the player. The console arrow keys move the player left, right, up, or down through the maze. Typing '0' stops the player. The player continues to move in the same direction until a wall of the maze stops him, you type a '0', or you type another arrow key.

When the player eats a power pellet, he acquires super power and can chase the ghosts briefly; the ghosts change color while the player has super power. If the player catches a ghost, he scores a bonus and the ghost returns to the ghost box temporarily. Once a player eats all the dots on the board, the game continues at the next level.

The upper left corner of the screen displays a score and the current board level. Each dot the player eats scores ten points. The first ghost a player eats while he has super power scores 200 points, the second 400, the third 800, and the fourth 1,600. At certain times during the game, a bonus letter appears below the ghost box; the player scores 100 points for eating the bonus letter on level 'A', 300 on level 'B', 500 on level 'C', and so on.

The lower left corner of the screen displays the number of extra players remaining in the current game (initally two). Another bonus player appears every 10,000 points, to a maximum of three extra players. The game ends when the ghosts eat the last player.

See Also

commands

chdir() — COHERENT System Call (libc)

Change working directory
chdir(*directory*) **char** *****_directory_**;**

The *working directory* (or *current directory*) is the directory from which the search for a file name begins if a path name does not begin with '/'. By convention, the working directory has the name '.'. **chdir** changes the working directory to the directory pointed to by *directory*. This change is in effect until the program exits or calls **chdir** again.

See Also

cd, chmod(), chroot(), COHERENT system calls, directory

Diagnostics

chdir returns zero if successful. It returns -1 if an error occurred, e.g., that *directory* does not exist, is not a directory, or is not searchable.

check — Command

Check file system
check [-s] *filesystem* ...

check uses the commands **icheck** and **dcheck** to check the consistency of a file system. It acts on each argument *filesystem* in turn; it calls first **icheck** and then **dcheck** on each to detect problems.

If **-s** is specified, **check** attempts to repair any errors automatically. You should first unmount the file system, if possible. If the root device is involved, you should be in single-user mode and then reboot the system immediately (without typing **sync**).

See Also

clri, commands, icheck, ncheck, sync, umount

Notes

Certain errors, such as duplicated blocks, cannot be fixed automatically. Decisions must be made by a human.

In earlier releases of COHERENT, **check** acted upon a default file system if none was specified.

This command has largely been superceded by **fsck**.

checklist — System Maintenance

File systems to check when booting COHERENT
/etc/checklist

The file **/etc/checklist** names all COHERENT partitions on your hard disk. COHERENT executes **fsck** for each file named in this file. This ensures that the file-system of each partition is checked and cleaned before it is mounted.

When you add a new COHERENT partition to your system, you should insert its name (that is, the name of its raw device) into **/etc/checklist** to ensure that its file system is checked at boot time.

See Also

mount.all, system maintenance

chgrp — Command

Change the group owner of a file
chgrp *group file* ...

chgrp changes the group owner of each *file* to *group*. The *group* may be specified by a valid group name or a valid numerical group identifier.

Only the superuser may use **chgrp**.

Files

/etc/group — Convert group name to group identifier

See Also

chmod, chown, commands

chmod() — COHERENT System Call (libc)

Change file-protection modes
#include <sys/stat.h>
chmod(*file*, *mode*)
char ***file*; **int** *mode*;

chmod sets the mode bits for *file*. The mode bits include protection bits, the set-user-id bit, and the sticky bit.

mode is constructed from the logical OR of the following, which are defined symbolically in the header file **stat.h**:

04000	Set user identifier
02000	Set group identifier
01000	Save file on swap device ("sticky bit")
00400	Read permission for owner
00200	Write permission for owner
00100	Execute permission for owner
00040	Read permission for members of owner's group
00020	Write permission for members of owner's group
00010	Execute permission for members of owner's group
00004	Read permission for other users
00002	Write permission for other users
00001	Execute permission for other users

For directories, some protection bits have a different meaning: write permission means files may be created and removed, whereas execute permission means that the directory may be searched.

The save-text bit (or "sticky bit") is a flag to the system when it executes a shared for of a load module. After the system runs the program, it leaves shared segments on the swap device to speed subsequent reinvocation of the program. Setting this bit is restricted to the superuser (to control depletion of swap space which might result from overuse).

Only the owner of a file or the superuser may change its mode.

See Also

COHERENT system calls, creat()

Diagnostics

chmod returns -1 for errors, such as *file* being nonexistent or the invoker being neither the owner nor the superuser.

chmod — Command

Change the modes of a file
chmod +*modes file*
chmod -*modes file*

The COHERENT system assigns a *mode* to every file, to govern how users access the file. The mode grants or denies permission to read, write, or execute a file.

The mode grants permission separately to the owner of a file, to users from the owner's group, and to all other users. For a directory, execute permission grants or denies the right to search the directory, whereas write permission grants or denies the right to create and remove files.

In addition, the mode contains three bits that perform special tasks: the set-user-id bit, the set-group-id bit, and the save-text or "sticky" bit. See the Lexicon entry for the COHERENT system call **chmod** for more information on how to use these bits.

The command **chmod** changes the permissions of each specified *file* according to the given *mode* argument. *mode* may be either an octal number or a symbolic mode. Only the owner of a *file* or the superuser may change a file's mode. Only the superuser may set the sticky bit.

A symbolic mode may have the following form. No spaces should separate the fields in the actual *mode* specification.

[which] how perm ... [, ...]

which specifies the permissions that are affected by the command. It may consist of one or more of the following:

- **a** All permissions, equivalent to **gou**
- **g** Group permissions
- **o** Other permissions
- **u** User permissions

If no *which* is given, **a** is assumed and **chmod** uses the file creation mask, as described in **umask**.

how specifies how the permissions will be changed. It can be

- **=** Set permissions
- **+** Add permissions
- **-** Take away permissions

perm specifies which permissions are changed. It may consist of one or more of the following:

g	Current group permissions
o	Current other permissions
r	Read permission
s	Setuid upon execution
t	Save text (sticky bit)
u	Current user permissions
w	Write permission
x	Execute permission

Multiple *how/perm* pairs have the same *which* applied to them. One or more specifications separated by commas tell **chmod** to apply each specification to the file successively.

The octal modes (see **stat**) are as follows:

04000	Set user id upon execution
02000	Set group id upon execution
01000	Sticky bit (save text)
00400	Owner read permission
00200	Owner write permission
00100	Owner execute permission
00040	Group read permission
00020	Group write permission
00010	Group execute permission
00004	Others read permission
00002	Others write permission
00001	Others execute permission

An octal *mode* argument to **chmod** is obtained by *or*ing the desired mode bits together.

Examples

The first example below sets the owner's permissions to read + write + execute, and the group and other permissions to read + execute. The second example adds execute permission for everyone.

```
chmod u=rwx,go=rx file
chmod +x file
```

See Also

chgrp, chown, commands, ls, stat, umask

chown() — COHERENT System Call (libc)

Change ownership of a file
chown(*file, uid, gid***)**
char ***file*;**
short *uid, gid*;

chown changes the owner of *file* to user id *uid* and group id *gid*.

To change only the user id without changing the group id, **stat** should be used to determine the value of *gid* to pass to **chown**.

chown is restricted to the superuser, because granting the ordinary user the ability to change the ownership of files might circumvent file space quotas or accounting based upon file ownership.

chown returns -1 for errors, such as nonexistent *file* or the caller not being the superuser.

See Also

chmod(), COHERENT system calls, passwd, stat()

chown — Command

Change the owner of files
chown *owner file* ...

chown changes the owner of each *file* to *owner*. The *owner* may be specified by valid user name or a valid numerical user id.

Only the superuser may use **chown.**

Files

/etc/passwd — To convert user name to user id

See Also

chgrp, chmod, commands

chroot() — COHERENT System Call

Change process's root directory
int chroot(*directory***)**
char ****directory*;**

The *root directory* is the directory from which file-name searches commence when a path name begins with '/'. **chroot** changes the root directory to *directory* for the requesting process and all of its children.

Because of security problems, **chroot** is restricted to the superuser. It is sometimes useful for a system administrator; for example, to test a new system environment that resides on a mounted file system.

See Also

chdir(), COHERENT system calls, fork()

Diagnostics

chroot returns zero for a successful call. It returns -1 on errors, such as the caller not being the superuser or the *directory* being nonexistent or not a directory.

ckermit — Command

Interactive inter-system communication and file transfer
ckermit [-abcdefghiklpqrstwx] [*file* ... **]**

ckermit implements the **kermit** communications protocol. It lets you communicate with other systems via modem or network, and to exchange files with other systems that have also implemented the **kermit** protocol. Unlike the **kermit** command also included with the COHERENT system, **ckermit** uses an interactive shell to remove some of the pain from the process of exchanging files. The name **ckermit** relects the fact that this command is written in the C language, and so has been ported to many different machines and operating systems.

You can run **ckermit** in either *interactive mode* or *command mode*. Simply typing the command

```
ckermit
```

invokes **ckermit** in interactive mode: **ckermit** displays a prompt, waits for your command, executes, then prompts you for its next command. Typing the command line plus one or more arguments invokes **ckermit** in command mode: **ckermit** then reads the arguments from the command line and executes them. After execution of the commands, **ckermit** returns to interactive mode.

ckermit's command-line options name either actions or settings. An action option tells **ckermit** to send a file, receive a file, or connect to a remote system. The command line may contain no more than one action option. A settings option changes one or more of the internal values that control how **ckermit** operates; for example, one setting option lets you set the baud rate of the serial port that **ckermit** will be using. A command line can contain any number of settings options.

Command-Line Options

ckermit recognizes the following command-line options:

-a *filename*
: Give an alternate name to a file being transferred. For example, the command

```
ckermit -s foo -a bar
```

transmits the file **foo** to a remote system, but tells the remote system that the file is named **bar**. Likewise, the command

```
ckermit -ra baz
```

stores the first incoming file under the name **baz**.

If more than one file arrives or is sent, only the first file is affected by the **-a** option.

-b *baudrate*
: Set the baud rate of the device to *baudrate*.

-c
: Connect to serial port, and pass all subsequent typing to that port To resume talking to your local system, type the escape character followed by the letter 'c'. The escape character is set by default to **<ctrl-\>**, although you can change it if you wish.

-d
: Debug mode — record debugging information in the file **debug.log** in the current directory.

-e *n*
: Set the length of the packet to *n* where *n* is a number between ten and about 1,000. Lengths of 95 or greater require that the implementation of **kermit** on the remote system support the long-packet extension to the **kermit** protocol.

-f
: Send a "finish" command to a remote server.

-g *file*
: Ask a remote system to send *file* or *files*. The file name must use the remote system's own syntax; you must quote all characters normally expanded by the COHERENT shell, e.g.:

```
ckermit -g x\*.\?
```

-h
: Help — display a brief synopsis of the command-line options.

-i
: The "image" option: specify that the file being transmitted or received is an eight-bit binary file, and therefore no conversion should be performed upon the data being received.

-k Passively receive file or files, copying them to standard output.

-l *device*
Name the serial device to be used. For example

```
ckermit -l /dev/com2
```

tells **ckermit** to use device **/dev/com2**.

-n Like **-c**, but used after a protocol transaction has occurred. You can use both **-c** and **-n** in the same command.

-p *x* Set parity, where *x* is one of **e**, **o**, **m**, **s**, or **n** (respectively, even, odd, mark, space, or none). If parity is other than none, then **ckermit** uses the eighth-bit prefixing mechanism to transfer binary data, provided the impementation of **kermit** on the remote system agrees. The default parity is none.

-q Quiet — suppress screen update during file transfer; for example, this lets you transfer a file in the background.

-r Receive a file or files. Wait passively for files to arrive.

-s *file* Send the specified *file* or *files*. If *fn* is '-' then **ckermit** sends from standard input, which may come from a file:

```
ckermit -s - < foo.bar
```

or come from a parallel process:

```
ls -l | ckermit -s -
```

You cannot use this mechanism to send text typed from the keyboard. To send a file named '-', precede it with a path name, e.g.:

```
ckermit -s ./-
```

-t Specify half duplex, line turnaround with XON as the handshake character.

-w Write-Protect — avoid file-name collisions for incoming files.

-x Begin server operation. This option can be used in either local or remote mode.

If **ckermit** is in local mode, shows the progress of the file transfer. A dot is printed for every four data packets; other packets are shown by type (e.g., 'S' for Send-Init); 'T' is printed when there's a timeout; and '%' is printed for each retransmission.

During file transfer, you can type the following "interrupt" commands:

<ctrl-F>
Interrupt the current file and go on to the next, if any.

<ctrl-B>
Interrupt the entire batch of files and terminate the transaction.

<ctrl-R>
Resend the current packet.

<ctrl-A>
Display a status report for the current transaction.

These interrupt characters differ from the ones used in other implementations of **ckermit** to avoid conflict with the COHERENT shell's interrupt characters.

Interactive Operation

When you invoke **ckermit** in interactive mode, it displays the following prompt.

```
C-Kermit>
```

Type any valid **ckermit** command; the set of valid commands is described below. **ckermit** executes the command and then prompts you for another. The process continues until you tell it to quit.

Commands begin with a keyword, normally an English verb, such as **send**. You can abbreviate any keyword, as long as you type enough characters to distinguish it from all other keywords. Certain commonly used keywords (e.g., **send**, **receive**, **connect**) **have special non-unique abbreviations (respectively, 's', 'r', and 'c').**

Certain characters have special functions in interactive commands:

? Print a message that explains what is possible or expected at the current point within a command. Depending upon the context, the message may be a brief phrase, a menu of keywords, or a list of files.

<esc> Request completion of the current keyword or file name, or insertion of a default value. **ckermit** will beep if the requested operation fails. **<tab>** does the same thing.

<del> Delete the previous character from the command. **<backspace>** does the same thing.

<ctrl-W>
Erase the rightmost word from the command line.

<ctrl-U>
Erase the entire command.

<ctrl-R>
Redisplay the current command.

<space>
Delimit fields (keywords, filenames, numbers) within a command.

<return>
Execute the command.

\ Insert any of the above characters into the command, literally. To enter a literal backslash, type two backslashes in a row (\\). Typing one backslash immediately **<return>** lets you continue the command on the next line.

ckermit recognizes the following interactive commands:

! *command*
Execute a shell command. A space must follow the **!**.

% A comment. **ckermit** ignores everything that follows the **%**.

bye Terminate and log out a remote **kermit** server.

close Close a log file.

connect
Connect to the remote system.

cwd *directory*
Change the working directory to *directory*.

dial Dial a telephone number.

directory
Display a directory listing.

echo Display arguments literally. Useful in take-command files.

exit Exit from the program, closing any open logs.

finish Instruct a remote **kermit** server to exit, but not log out.

get Get files from a remote **kermit** server.

hangup
Hang up the telephone.

help Display a help message for a given command.

log Open a log file — debugging, packet, session, transaction.

quit Same as **exit**.

receive Passively wait for files to arrive.

remote Issue file-management commands to a remote **kermit** server.

script Execute a login script with a remote system.

send *file*
Send *file* to the remote **kermit** server.

server Begin server operation.

set Set various internal parameters.

show Display values of parameters, program version, etc.

space Display current disk space usage.

statistics
Display statistics about most recent transaction.

take Execute commands from a file.

Interactive **ckermit** accepts commands from files as well as from the keyboard. Upon startup, **ckermit** looks for the file **.kermrc** first in directory **$HOME** and then in the current directory; if it finds the file, it executes all commands it finds therein. These commands must be in interactive format. Command files may be nested to any reasonable depth.

The set Command

As noted above, the **set** command lets you set the internal parameters by which **ckermit** operates. The **set** command recognizes the following arguments:

block-check
Level of packet error detection.

delay Time to wait before sending first packet.

duplex Specify which side echoes during connect mode.

escape-character
Character to prefix *escape commands* during connect mode.

file Set various file parameters.

flow-control
Communication line full-duplex flow control.

handshake
Communication line half-duplex turnaround character.

line Communication-line device name.

modem-dialer
Type of modem-dialer on communication line.

parity Communication line character parity.

prompt
Change the **ckermit** program's prompt.

receive Set various parameters for inbound packets.

retry Set the packet retransmission limit.

send Set various parameters for outbound packets.

speed Communication line speed.

Remote Commands

ckermit also has a suite of commands that are sent to the remote system for execution. They are as follows:

cwd Change remote working directory (also, **remote cd**).

delete Delete remote files.

directory
Display a listing of remote file names.

help Request help from a remote server.

host Issue a command to the remote host in its own command language.

space Display current disk space usage on remote system.

type Display a remote file on your screen.

who Display the users logged in to the remote system, or get information about a user.

Files

.kermrc — **ckermit** initialization commands

See Also

commands, kermit, uucp

Notes

The **kermit** protocol was developed at the Columbia University Center for Computing Activities.

clear — Command

Clear the screen
clear

The command **clear** reads the **termcap** description of your terminal and uses the information therein to clear your terminal's screen. The environmental variable **TERM** must define your terminal's type.

See Also

commands, TERM, termcap

clearerr() — STDIO Macro (stdio.h)

Present stream status
#include <stdio.h>
clearerr(*fp***) FILE ***fp***;**

clearerr resets the error flag of the argument *fp*. If an error condition is detected by the related macro **ferror**, **clearerr** can be called to clear it.

Example

For an example of this function, see the entry for **ferror**.

See Also

ferror(), STDIO

close() — COHERENT System Call (libc)

Close a file
int close(*fd***) int** *fd***;**

close closes the file identified by the file descriptor *fd*, which was returned by **creat**, **dup**, **open**, or **pipe**. **close** also frees the associated file descriptor.

Because each program can have only a limited number of files open at any given time, programs that process many files should **close** files whenever possible. The function **exit** automatically calls **fclose** for all open files; however, the system call **_exit** does not.

Example

For an example of this function, see the entry for **open**.

See Also

COHERENT system calls, creat(), open()

Diagnostics

close returns -1 if an error occurs, such as its being handed a bad file descriptor; otherwise, it returns zero.

clri — Command

Clear i-node
/etc/clri *filesystem inumber* ...

clri zeroes out each i-node with a given *inumber* on *filesystem*. *filesystem* is almost always a device-special file that corresponds to a disk device. The raw device should be used.

The user must have read and write permission on the *filesystem*. If the *inumber* corresponds to an open file, the **clri** has a very high probability of being ineffective: the system maintains in core memory a copy of all active i-nodes, and this copy will eventually be written out to

disk, undoing the effects of **clri**. To counter this problem, unmount the file system before running **clri**. If the i-node is for the root file system, you must reboot the system immediately after running **clri**.

See Also

commands, dcheck, fsck, icheck, i-node, umount

cmp — Command

Compare bytes of two files
cmp [-ls] *file1 file2* [*skip1 skip2*]

cmp is a command that is included with COHERENT. It compares *file1* and *file2* character by character for equality. If *file1* is '-', **cmp** reads the standard input.

Normally, **cmp** notes the first difference and prints the line and character position, relative to any skips. If it encounters EOF on one file but not on the other, it prints the message "EOF on file*n*". The following are the options that can be used with **cmp**:

-l Note each differing byte by printing the positions and octal values of the bytes of each file.

-s Print nothing, but return the exit status.

If the skip counts are present, **cmp** reads *skip1* bytes on *file1* and *skip2* bytes on *file2* before it begins to compare the two files.

See Also

commands, diff, sh

Diagnostics

The exit status is zero for identical files, one for non-identical files, and two for errors, e.g., bad command usage or inaccessible file.

COHERENT system calls — Overview

The COHERENT system makes many services available to the C programmer. A programmer can use a COHERENT service through a system call. COHERENT's libraries include the following system calls:

_exit() . . . . . Terminate a process
access() . . . . . Check if file can be accessed in given mode
acct() . . . . . Enable/disable process accounting
alarm() . . . . . Set a timer
brk() . . . . . Change size of data area
chdir() . . . . . Change working directory
chmod() . . . . . Change file protection modes
chown() . . . . . Change ownership of a file
chroot() . . . . . Change process's root directory
close() . . . . . Close a file
creat() . . . . . Create/truncate a file
dup() . . . . . Duplicate a file descriptor
dup2() . . . . . Duplicate a file descriptor
execl() . . . . . Execute a load module
execle() . . . . . Execute a load module
execlp() . . . . . Execute a load module
execv() . . . . . Execute a load module
execve() . . . . . Execute a load module
execvp() . . . . . Execute a load module

fcntl() . . . . . Manipulate an open file
fork() . . . . . Create a new process
fstat() . . . . . Find file attributes
getegid() . . . . . Get effective group id
geteuid() . . . . . Get effective user id
getgid() . . . . . Get real group id
getpid() . . . . . Get process id
getuid() . . . . . Get real user id
gtty() . . . . . Terminal initialization
ioctl() . . . . . Device-dependent control
kill() . . . . . Send a signal to a process
link() . . . . . Create a link
lseek() . . . . . Set read/write position
mknod() . . . . . Create a special file
mount() . . . . . Mount a file system
msgctl() . . . . . Control message operation
msgget() . . . . . Get a message queue
msgrcv() . . . . . Receive a message
msgsnd() . . . . . Send a message
open() . . . . . Open a file
pause() . . . . . Wait for signal
pipe() . . . . . Create a pipe
ptrace() . . . . . Trace process execution
read() . . . . . Read from a file
sbrk() . . . . . Increase a program's data space
semctl() . . . . . Control semaphore operations
semget() . . . . . Get a set of semaphores
semop() . . . . . Perform semaphore operations
setgid() . . . . . Set group id and user id
setuid() . . . . . Set user id
shmctl() . . . . . Control shared-memory operations
shmget() . . . . . Get the shared-memory segment
signal() . . . . . Specify disposition of a signal
sload() . . . . . Load device driver
stat() . . . . . Find file attributes
stime() . . . . . Set the time
stty() . . . . . Set terminal modes
suload() . . . . . Unload device driver
sync() . . . . . Flush system buffers
times() . . . . . Obtain process execution times
umask() . . . . . Set file creation mask
umount() . . . . . Unmount a file system
unlink() . . . . . Remove a file
utime() . . . . . Change file access and modification times
wait() . . . . . Await completion of child process
write() . . . . . Write to a file

See Also

Lexicon, libraries, STDIO

col — Command

Remove reverse and half-line motions
col [**-bdfx**][**-p***n*]

The command **col** reads the standard input and writes to the standard output. It removes reverse and half-line motions from the output of **nroff** for the benefit of output devices that cannot perform them. It maintains an image of the page in memory and performs these motions virtually so they do not appear on the output.

col understands four escape sequences: **<esc> 7** for reverse line feed, **<esc> 8** for half reverse line feed, **<esc> 9** for half forward line feed, and **<esc> B** for a forward line feed. It removes **<esc>** (ASCII 033) from the input stream if it is followed by any other character.

Eight control characters besides **<esc>** are interpreted by **col**. Newline, return, space, backspace, and tab carry their usual meaning. **VT** (013) is an alternate form of reverse line feed. The characters **SO** (017) and **SI** (016) signal the start and end of text in an alternate character set. **col** remembers the character set for each character and uses **SO** and **SI** to distinguish them on the output. **col** removes all other control characters from the input stream.

col recognizes the following options:

-b The output device cannot backspace. Only the last of a set of characters destined for a given position will appear.

-d Double-space the output. This doubles the length of a document but preserves relative vertical spacing. The **-f** option has precedence.

-f The output device can perform half-forward line feeds. Full lines appear single spaced with half lines between them. This is the only situation in which half forward line feeds appear in the output of **col** — reverse line motions never appear.

-x Suppress the default conversion of white space to tabs on output.

-p *n* Set the internal page buffer size to *n* full lines (default, 128).

If neither **-f** nor **-d** is chosen, **col** moves non-empty half lines to the next lower full line and pushes all later lines down one line. This can distort the appearance of the document.

See Also

ASCII, commands, nroff

Notes

Backing up past the start of a document or of the page buffer loses characters.

com — Device Driver

Device drivers for asynchronous serial lines

The COHERENT system has drivers for four asynchronous serial lines, **com1** through **com4**.

A serial line can be opened into any of four different "flavors", as follows:

com?l	Interrupt driven, local mode (no modem control)
com?r	Interrupt driven, remote mode (modem control)
com?pl	Polled, local mode (no modem control)
com?pr	Polled, remote mode (modem control)

"Local mode" means that the line will have a terminal plugged into it, to directly access the computer. "Modem control" means that the line will have a modem plugged into it. Modem control is enabled on a serial line by resetting the modem control bit (bit 7) in the minor number for the device. This allows the system to generate a hangup signal when the modem indicates loss of carrier by dropping DCD (Data Carrier Detect). A modem line should always have its DSR, DCD and CTS pins connected. If left hanging, spurious transitions can cause severe system thrashing. To disable modem control on a given serial line, use the minor device which has the modem control bit set (bit 7). An **open** to a modem-control line will block until a carrier is detected (DCD goes true).

"Interrupt mode" means that the port can generate an interrupt to attract the attention of the COHERENT system; "polled mode" means that the port cannot generate an interrupt, but must be checked (or "polled") constantly by the COHERENT system to see if activity has occurred on it.

The COHERENT system uses two device drivers to manage serial lines: one driver manages COM1 and COM3, and the other manages COM2 and COM4. Due to limitations in the design of the ports, you can enable interrupts on either COM1 or COM3 (or on COM2 or COM4), but not both. If you wish to use both ports simultaneously, one must be run in polled mode. For example, if you wish to open all four serial lines, you can open two of the lines in interrupt mode: you can open either COM1 or COM3 in interrupt mode, and you can open either COM2 or COM4 in interrupt mode. The other two lines must be opened in polled mode.

Opening a device in polled mode consumes many CPU cycles, based upon the speed of the highest baud rate requested. For example, on a 20 MHz 80386-based machine, polling at 9600-baud was found to consume about 15% of the CPU time. As only one device can use the interrupt line at any given time, the best approach is to make the high-speed line of the pair interrupt driven and open the low-speed or less-frequently used line in polled mode. However, if you enable a polled line for logins, the port is open and will be polled as long as the port remains open (enabled). Thus, even if a port is not in use, the fact that it has a **getty** on it consumes CPU cycles. As a rule of thumb, try and open a port in interrupt mode. If you cannot, use the polled version. Also note that use of any of the four serial ports in polled mode prevents other polled serial device drivers, such as the **hs** generic multi-port polled serial driver, from being used at the same time.

If you intend to use a modem on your serial port, you must insure that the DCD signal from the modem actually *follows* the state of carrier detect. Some modems allow the user to "strap" or set the DCD signal so that it is always asserted (true). This incorrect setup will cause COHERENT to think that the modem is "connected" to a remote modem, even when there is no such connection.

In addition, if you wish to allow remote logins to your COHERENT system via your modem, you must insure that the modem does **not** echo any commands or status information. Failure to do so will result in severe system thrashing due to the **getty** or **login** processes endlessly "talking" to your modem.

Changing Default Port Speeds

Serial lines **com1** through **com4** default to 9600 baud when opened. This default speed can be permanently changed on a "per port" basis by changing the value of driver variables **C1BAUD_**, **C2BAUD_**, **C3BAUD_** or **C4BAUD_**. The list of acceptible values can be found in header file **<sgtty.h>** and range from 1, corresponding to 50 baud, up to 17, which corresponds to 19,200 baud. For a table of legal baud rates, see the Lexicon entry for **sgtty.h**.

To change the default value for a port, you must use the **/conf/patch** command. For

example, to change the default speed for port **com2** to 2400 baud, enter the following command while running as the superuser:

```
/conf/patch /drv/all C2BAUD_=12
```

The change will not take effect until the next time that you boot your system.

See Also

com1, com2, com3, com4, device drivers

Diagnostics

An attempt to open a non-existent device will generate error messages. This can occur if hardware is absent or not turned on.

Notes

The **com*** series of devices are not compatible with the **ioctl()** parameters defined in header file **<termio.h>**. Be sure to include header file **<sgtty.h>** if you wish to perform terminal specific **ioctl()** calls.

In the current version of these drivers, the following sequence of steps results in a panic:

```
enable com4pl
enable com3pl
disable com4pl
kill kill <all driver process id>
```

The key is that the driver containing the polling routine cannot be unloaded if the other driver is still polling.

Note, too, that if any **com** device driver is used in polling mode, the **hs** driver cannot be used, and vice versa.

com1 — Device Driver

Device driver for asynchronous serial line COM1

/dev/com1 is the COHERENT system's standard interface to asynchronous serial line COM1. The interface is assigned major device 5, and is accessed as a character-special device. The I/O address for the corresponding 8250 SIO is 0x3F8 (COM1). **com1** generates interrupt IRQ4.

Four versions of device **com1** are in directory **/dev**, as follows:

Device Name	*Major*	*Minor*	*I/O Type*	*Modem Control?*
/dev/com1l	5	128	Interrupts	No
/dev/com1r	5	0	Interrupts	Yes
/dev/com1pl	5	192	Polled	No
/dev/com1pr	5	64	Polled	Yes

For details on how these versions differ, see the entry for **com**.

Files

/dev/com1l — Interrupt-driven, non-modem (local) line
/dev/com1r — Interrupt-driven, modem (non-local) line
/dev/com1pl — Polled, non-modem (local) line
/dev/com1pr — Polled, modem (non-local) line

See Also

com, com3, stty

com2 — Device Driver

Device driver for asynchronous serial line COM2

/dev/com2 is the COHERENT system's standard interface to asynchronous serial line COM2. The interface is assigned major device 6, and is accessed as a character-special device. The I/O address for the corresponding 8250 SIO is 0x2F8 (COM2). **com2** generates interrupt IRQ3.

Four versions of device **com2** are in directory **/dev**, as follows:

Device Name	*Major*	*Minor*	*I/O Type*	*Modem Control?*
/dev/com2l	6	128	Interrupts	No
/dev/com2r	6	0	Interrupts	Yes
/dev/com2pl	6	192	Polled	No
/dev/com2pr	6	64	Polled	Yes

For details on how these differ, see the entry for **com**.

Files

/dev/com2l — Interrupt-driven, non-modem (local) line
/dev/com2r — Interrupt-driven, modem (non-local) line
/dev/com2pl — Polled, non-modem (local) line
/dev/com2pr — Polled, modem (non-local) line

See Also

com, com4, stty

com3 — Device Driver

Device driver for asynchronous serial line COM3

/dev/com3 is the COHERENT system's standard interface to asynchronous serial line COM3. The interface is assigned major device 5, and is accessed as a character-special device. The I/O address for the corresponding 8250 SIO is 0x3E8 (COM3). **com3** generates interrupt IRQ4.

Four versions of device **com3** are in directory **/dev**, as follows:

Device Name	*Major*	*Minor*	*I/O Type*	*Modem Control?*
/dev/com3l	5	129	Interrupts	No
/dev/com3r	5	1	Interrupts	Yes
/dev/com3pl	5	193	Polled	No
/dev/com3pr	5	65	Polled	Yes

For details on how these differ, see the entry for **com**.

Files

/dev/com3l — Interrupt-driven, non-modem (local) line
/dev/com3r — Interrupt-driven, modem (non-local) line
/dev/com3pl — Polled, non-modem (local) line
/dev/com3pr — Polled, modem (non-local) line

See Also

com, com1, stty

com4 — Device Driver

Device driver for asynchronous serial line COM4

/dev/com4 is the COHERENT system's standard interface to asynchronous serial line COM4. The interface is assigned major device 6, and is accessed as a character-special device. The I/O address for the corresponding 8250 SIO is 0x2E8 (COM4). **com4** generates interrupt IRQ3.

Four versions of device **com4** are in directory **/dev**, as follows:

Device Name	*Major*	*Minor*	*I/O Type*	*Modem Control?*
/dev/com4l	6	129	Interrupts	No
/dev/com4r	6	1	Interrupts	Yes
/dev/com4pl	6	193	Polled	No
/dev/com4pr	6	65	Polled	Yes

For details on how these differ, see the entry for **com**.

Files

/dev/com4l — Interrupt-driven, non-modem (local) line
/dev/com4r — Interrupt-driven, modem (non-local) line
/dev/com4pl — Polled, non-modem (local) line
/dev/com4pr — Polled, modem (non-local) line

See Also

com, com2, stty

comm — Command

Print common lines
comm [-123] *file1 file2*

The command **comm** prints the lines unique to *file1* in the first column, the lines unique to *file2* in the second column, and the lines common to both in the third. Both *file1* and *file2* should be sorted in ASCII order. Any or all columns may be suppressed by indicating the column or columns to suppress in the optional flag. The file '-' means standard input.

See Also

cmp, commands, diff, sort, uniq

commands — Overview

The following lists the commands included with COHERENT. The command name is given on the left and a description on the right.

Communications

The following commands let you exchange information with other users and other systems.

ckermit . . . . . Interactive inter-system communication and file transfer
kermit . . . . . Remote system communication and file transfer
mail. . . . . Send/read electronic mail
mesg . . . . . Permit/deny messages from other users
msg . . . . . Send a brief message to other users

msgs . . . . . Read messages intended for all COHERENT users
uucico . . . . . Connect to a remote system
uucp . . . . . Copy a file to or from a remote system
wall . . . . . Send a message to all logged in users
write . . . . . Converse with another user

Device Handling

The following commands help you run peripheral devices, especially printers. For commands that drive communications devices, e.g., modems, see the section on *Communications*, above.

epson . . . . . Print a file on an Epson printer
fnkey . . . . . Set/print function keys for a terminal
hp . . . . . Prepare files for HP LaserJet-compatible printer
hpr . . . . . Send to LaserJet printer spooler
hpskip . . . . . Abort/restart current listing on LaserJet
lpr . . . . . Send to line printer spooler
lpskip . . . . . Terminate/restart current line printer listing
stty . . . . . Set/print terminal modes
tty . . . . . Print the user's terminal name
ttystat . . . . . Get terminal status

Directory and File Handling

The following commands let you create, remove, and otherwise manipulate files and directories.

cat . . . . . Concatenate/print files
cd . . . . . Change directory
chgrp . . . . . Change the group owner of a file
chmod . . . . . Change the modes of a file
chown . . . . . Change ownership of a file
cmp . . . . . Compare bytes of two files
compress . . . . . Compress a file
cp . . . . . Copy a file
cpdir . . . . . Copy directory hierarchy
dd . . . . . File conversion
dos . . . . . Transfer files to/from an MS-DOS file system
fdisk . . . . . View/change hard-disk partitioning
file . . . . . Name a file's type
find . . . . . Search for files satisfying a pattern
lc . . . . . List/categorize files in a directory
ln . . . . . Create a link to a file
ls . . . . . List directory's contents
mkdir . . . . . Create a directory
mv . . . . . Rename files or directories
pwd . . . . . Print the name of the current directory
qfind . . . . . Find all files with a given name
rm . . . . . Remove files
rmdir . . . . . Remove directories
touch . . . . . Update modification time of a file
uncompress . . . . . Uncompress a file
whereis . . . . . Locate source, binary, and manual files
which . . . . . Locate executable files
zcat . . . . . Concatenate a compressed file

Editors

COHERENT includes a number of text editors, to suit a variety of tastes.

ed . . . . . Interactive line editor
elvis. . . . . Berkeley-style screen editor
ex . . . . . Berkeley-style line editor
me. . . . . MicroEMACS screen editor
sed . . . . . Stream editor
vi . . . . . Berkeley-style screen editor

Games

The following commands are just for fun.

banner . . . . . Print large sized letters
cal. . . . . Print a calendar
chase . . . . . Highly amusing video game
fortune. . . . . Print randomly selected, hopefully humorous, text
lines . . . . . Highly amusing board game

Languages and Programming Tools

The COHERENT system comes with a number of languages, and a set of tools for maintaining and debugging your programs.

as . . . . . Mark Williams assembler
awk . . . . . Report generation, pattern scanning, and processing language
cc . . . . . C-language compiler
conv . . . . . Numeric base converter
cpp . . . . . C preprocessor
db . . . . . Assembler-level symbolic debugger
fixstack . . . . . Alter size of a program's stack
ld . . . . . Link relocatable object files
lex. . . . . Lexical analyzer generator
m4 . . . . . Macro processor
make . . . . . Program building discipline
nm . . . . . Print a program's symbol table
od . . . . . Print an octal dump of a file
prof . . . . . Print execution profile of a C program
ref. . . . . Display a C function header
srcpath. . . . . Find source files
size . . . . . Print size of an object file
strip . . . . . Strip symbol tables from executable file
yacc. . . . . Parser generator

Libraries and Archives

The following commands help you create and read libraries and archives. These can be used as libraries (such as the libraries used when linking a C program), or to back up files.

ar . . . . . The object librarian/archiver
cpio . . . . . Archiving/backup utility
dump . . . . . File-system backup utility
dumpdate . . . . . Print dump dates
dumpdir . . . . . Print the directory of a dump
pax . . . . . Portable archive interchange
ranlib . . . . . Create index for object library
restor. . . . . Restore file system
tar. . . . . V7 tape archive manager

ustar . . . . . Tape archive utility

Mail

COHERENT comes with with a full-featured, UNIX-style mail facility. This is described in the overview article **mail**. The following commands perform mail-related work; note that some are also listed in other sections of this article.

mail. . . . . Send/read electronic mail
mkfnames . . . . . Generate data base of user names
nptx. . . . . Generate permutations of users' full names

Shell Commands

COHERENT comes with two command interpreters, or *shells*: **ksh**, the Korn shell, and **sh**, the Bourne shell. The following commands are used either by the Korn shell, by the Bourne shell, or by both.

alias. . . . . Set an alias
basename . . . . . Strip path information from a file name
bind. . . . . Bind key sequence to editing command
break . . . . . Exit from shell construct
builtin . . . . . Execute a command as a built-in command
case. . . . . Execute commands conditionally according to pattern
continue. . . . . Terminate current iteration of shell construct
dirs . . . . . Print contents of directory stack
echo . . . . . Repeat an argument
eval . . . . . Evaluate arguments
exec. . . . . Execute command directly
exit . . . . . Exit from a noninteractive shell
export . . . . . Add a shell variable to the environment
expr. . . . . Compute a command line expression
false. . . . . Unconditional failure
fc . . . . . Edit and re-execute one or more previous commands
for. . . . . Execute commands for tokens in list
from . . . . . Generate list of numbers, for use in loop
getopts. . . . . Parse command-line options
hash . . . . . Add a command to the shell's hash table
if. . . . . Execute a command conditionally
jobs . . . . . Print information about stopped jobs
let . . . . . Evaluate an expression
popd . . . . . Pop an item from the directory stack
prep. . . . . Produce a word list
print . . . . . Echo text onto the standard output
pushd. . . . . Push an item onto the directory stack
read. . . . . Assign values to shell variables
readonly . . . . . Mark a shell variable as read only
set. . . . . Set shell option flags and positional parameters
shift. . . . . Shift positional parameters
sleep . . . . . Stop executing for a specified time
tee. . . . . Branch pipe output
test . . . . . Evaluate conditional expression
times . . . . . Print total user and system times
trap . . . . . Execute command on receipt of signal
true . . . . . Unconditional success
typeset. . . . . Set/list variables and their attributes
umask . . . . . Set the file-creation mask
unalias . . . . . Remove an alias

until . . . . . Execute commands repeatedly
wait . . . . . Await completion of background process
whence. . . . . List a command's type
while . . . . . Execute commands repeatedly

String Processing

Some of the most useful commands are those that process strings. COHERENT has many commands that search for strings, manipulate strings, sort strings, and otherwise perform useful manipulations on strings.

c . . . . . Print multi-column output
cgrep . . . . . Pattern search for C programs
comm. . . . . Print common lines
cut . . . . . Select portions of each line of a file
detab . . . . . Replace tab characters with spaces
diff . . . . . Summarize differences between two files
diff3. . . . . Summarize differences among three files
egrep . . . . . Extended pattern search
grep. . . . . Pattern search
head . . . . . Print the beginning of a file
join . . . . . Join two data bases
look. . . . . Find matching lines in a sorted file
more . . . . . Display text one screenful at a time
paste . . . . . Merge lines of files
rev . . . . . Print text backwards
scat . . . . . Print text files one screenful at a time
sort . . . . . Sort lines of text
split. . . . . Split a text file into smaller files
strings . . . . . Print all character strings from a file
tail . . . . . Print the end of a file
tr . . . . . Translate characters
tsort . . . . . Topological sort
uniq. . . . . Remove/count repeated lines in a sorted file
view. . . . . Berkeley-style text viewer
wc . . . . . Count words, lines, and characters in text files

System Accounting

The following commands help you to keep track of how your COHERENT system is working.

ac . . . . . Summarize login accounting information
accton . . . . . Enable/disable process accounting
df . . . . . Measure free space on disk
du . . . . . Summarize disk usage
ps . . . . . Print process status
sa . . . . . Print a summary of process accounting
quot. . . . . Summarize file-system usage
time. . . . . Time the execution of a command
times . . . . . Print total user and system times
uulog . . . . . Examine UUCP operations

System Maintenance

These commands help you to maintain your COHERENT system.

at . . . . . Execute commands at given time
bad . . . . . Maintain list of bad blocks
badscan . . . . . Examine a device for bad blocks

build . . . . . Install COHERENT onto a hard disk
check. . . . . Check file system
clri . . . . . Clear i-node
date. . . . . Print/set the date and time
dcheck . . . . . Check directory consistency
drvld . . . . . Load loadable drivers into memory
fdformat . . . . . Format a floppy disk
fsck . . . . . Check and repair file systems interactively
icheck . . . . . i-node consistency check
man. . . . . Print Lexicon entries
mkfs . . . . . Make a new file system
mknod . . . . . Make a special file or named pipe
mount . . . . . Mount a file system
ncheck . . . . . Print file names corresponding to i-node
newgrp . . . . . Change to a new group
newusr . . . . . Add new user to COHERENT system
reboot . . . . . Reboot the COHERENT system
shutdown . . . . . Shut down the COHERENT system
sync. . . . . Flush system buffers
umount . . . . . Set file creation mask
unmkfs. . . . . Create a prototype file system

Text Processors

These commands help you to create orderly, attractive printed text. For information on how to print the output of these commands, see the commands listed under *Device Handling*, above.

col. . . . . Remove reverse and half line motions
deroff. . . . . Remove text formatting control information
nroff . . . . . Text-formatting language
fwtable . . . . . Build a font-width table from PCL or PostScript font
pr . . . . . Paginate and print files
prps. . . . . Paginate and print files on PostScript printers
spell. . . . . Find spelling errors
troff. . . . . Extended text-formatting language
typo. . . . . Detect possible typographical and spelling errors

UUCP

The UUCP commands lets you form a network with other COHERENT or UNIX systems. Members of the network can grant each other permission to exchange mail and execute commands on each others' systems remotely and automatically, without having to be directed by a human being. The overview article **UUCP** describes the COHERENT UUCP facility in some detail. The following commands perform UUCP-related work; note that some of the commands listed here also are also listed in other sections of this article.

uucico . . . . . Connect to a remote system
uucp . . . . . Copy a file to or from a remote system
uudecode . . . . . Decode a transmitted UUCP file
uuencode . . . . . Encode a UUCP file for tranmission
uuinstall. . . . . Configure UUCP control files
uulog . . . . . Examine UUCP operations
uumvlog . . . . . Archive UUCP log files
uuname . . . . . Print names of recognized systems
uurmlock . . . . . Remove UUCP lock files
uutouch . . . . . Force polling of a remote site
uux . . . . . Execute a command on a remote system

uuxqt . . . . . Execute file as requested by remote system

Miscellaneous

The following commands do not fit neatly into any of the above categories. These include some of the more interesting and useful COHERENT commands, and are worth your attention.

ATclock . . . . . Read/set the AT realtime clock
bc . . . . . Interactive calculator with arbitrary precision
calendar . . . . . Electronic reminder service
clear . . . . . Clear your terminal's screen
crypt . . . . . Encrypt/decrypt text
dc . . . . . Desk calculator
disable . . . . . Disable a port
enable . . . . . Enable a port
factor. . . . . Factor a number
help . . . . . Print concise description of command
install . . . . . Install a software update onto COHERENT
kill . . . . . Signal a process
ksh . . . . . Invoke the Korn shell
login . . . . . Log in or change user name
passwd . . . . . Set/change login password
phone. . . . . Print numbers and addresses from phone directory
sh . . . . . Invoke the Bourne shell
su . . . . . Substitute user id, become superuser
sum . . . . . Print checksum of a file
units . . . . . Convert measurements
who . . . . . Print who is logged in
yes . . . . . Print infinitely many responses

For more information on any of these commands, see its entry within the Lexicon.

See Also

Lexicon

compress — Command

Compress a file
compress [**-dfvc**] [**-b***num*] [**-w** *tmpfile*] [*file* ...]

compress compresses a file using the Lempel-Ziv algorithm. With text files and archives, it often can achieve 50% rate of compression.

If one or more *files* are specified on the command, **compress** compresses them and appends the suffix **.Z** onto the end of each compressed file's name. If no *file* is specified on the command line, **compress** compresses text from the standard input and writes the compressed text to the standard output.

compress recognizes the following options:

-d Decompress rather than compress.

-f Force an output file to be generated even if no space is saved by compression.

-v Verbose mode: force **compress** to write statistics about its performance.

-c Send output to stdout.

-b The "bits" option. **compress** uses the compression level set via the *num* argument. Previous releases of **compress** would only allow values of *num* up to 12, with 12 being the default value if the **-b** option was not specified. The version of **compress** introduced with COHERENT version 3.1 handles values up to 16, with 12 being the default.

-w The "workfile" option. **compress** uses *tmpfile* to write its temporary file. By default **compress** uses RAM device **/dev/ram1** for temporary storage. For this reason, it is strongly advised that you not use **/dev/ram1** as a RAM disk. This option may be necessary on machines with limited amounts of RAM.

If you wish to ensure backwards compatibility with previous releases of COHERENT, do not use **compress** with a *num* value greater than 12.

See Also

commands, ram, uncompress, zcat

con.h — Header File

Configure device drivers
#include <sys/con.h>

The header file **con.h** gives the configuration for each device driver included with the COHERENT system. Each driver is defined using the structure **CON**, which is declared in **<sys/con.h>**.

See Also

header files, sload()

console — Device Driver

Console device driver

/dev/console is the device driver for the console of a COHERENT system on the IBM AT. It is assigned major device number 2 and minor device number 0.

/dev/console interprets escape sequences in console output to control output on the console monitor. These escape sequences are compatible with ANSI X3.25. Thus, they are similar to those used by the DEC VT-100 and VT-220 terminals.

The special sequences include the following:

<esc>>= Enter alternate keypad mode.

<esc>> Exit alternate keypad mode.

<esc>n Print the corresponding special graphics character.

<esc>7 Save the current cursor position.

<esc>8 Restore the previously saved cursor position.

<esc>c Reset to power-up configuration

<esc>D Move the cursor down one line without changing the column position. This command moves the scrolling region text up and inserts blank lines if required.

<esc>E Move the cursor to the first column of the next line. This command move the scrolling region down and inserts blank line if required.

<esc>M	Move the cursor up one line without changing column position
<esc>[A	Cursor up; stop at top of page.
<esc>[B	Cursor down; stop at bottom edge of scrolling region.
<esc>[C	Cursor right. Stop at right bottom corner of scrolling region.
<esc>[D	Cursor left.
<esc>[E	Cursor next line. Move scrolling region up and insert a blank line if required.
<esc>[F	Move scrolling region text down and insert a blank line if required.
<esc>[*n* **G**	Move the cursor to the *n*th column of the current line.
<esc>[*n*;*m* **H**	Move the cursor to position *m n*. Position is relative to the scrolling region.
<esc>[I	Move the cursor position to the next horizontal tabulation stop.
<esc>[*n* **J**	Erase display: **0** Erase from cursor to end of screen. **1** Erase from beginning of screen to cursor. **2** Erase the entire screen.
<esc>[*n* **K**	Erase line: **0** Erase from cursor to end of line. **1** Erase from beginning of line to cursor. **2** Erase entire line.
<esc>[L	Insert a line.
<esc>[M	Delete a line.
<esc>[*n* **O**	Erase scrolling region: **0** Erase from cursor to end of scrolling region. **1** Erase from beginning of scrolling region to cursor. **2** Erase entire scrolling region. Reposition cursor at top left corner of scrolling region.
<esc>[S	Scroll the characters in the scrolling region up one line. The bottom of the scrolling region is cleared to blanks.
<esc>[T	Scroll the characters in the scrolling region down one line. The top line of the scrolling region is cleared to blanks.
<esc>[Z	Move the cursor backwards to the last tabulation stop.
<esc>[*n* **'**	Move the cursor to column *n* of the current line.
<esc>[*n* **a**	Move the cursor forward n columns in the current line.
<esc>[*n* **d**	Move the cursor to row *n* of the display.
<esc>[*n* **e**	Move the cursor down *n* rows.
<esc>[*n*;*m* **f**	Move the cursor to column *m* of row *n* in the display.

<esc>[*n;m* **g** Position cursor to column *m* of line *n*. Positioning is relative to the scrolling region.

<esc>[*n* **m** Select graphics rendition:

0	All attributes off.
1	Bold intensity.
4	Underscore on.
5	Blink on.
7	Reverse video.
30	Black foreground.
31	Red foreground.
32	Green foreground.
33	Brown foreground.
34	Blue foreground.
35	Magenta foreground.
36	Cyan foreground.
37	White foreground.
40	Black background.
41	Red background.
42	Green background.
43	Brown background.
44	Blue background.
45	Magenta background.
46	Cyan background.
47	White background.
50	Black border.
51	Red border.
52	Green border.
53	Brown border.
54	Blue border.
55	Magenta border.
56	Cyan border.
57	White border.

<esc>[*n;m* **r** Display lines *n* through *m* become the scrolling region.

<esc>[*n* **v** Select cursor rendition:

0	Cursor visible.
1	Cursor invisible.

<esc>[?4h Enable smooth scrolling. This eliminates snow at the expense of speed.

<esc>[?4l Disable smooth scrolling (default).

<esc>[?7h Enable wraparound. Typing past column 80 moves the cursor to the first column of the next line, scrolling if necessary.

<esc>[?7l Disable wraparound. The cursor will not move past column 80. This is useful if the screen is being used as a block mode interface.

<esc>' Disable manual input. Terminal "beeps" (outputs **<ctrl-G>**) when a key is typed on the keyboard. Interrupt and quit signals are still passed to the terminal process. Input may be renabled via **<esc>c** (power up reset) or **<esc>b** (enable manual input).

<esc>b Enable keyboard input that has been disabled by **<esc>'** (disable manual input).

<esc>t Enter keypad-shifted mode.

<esc>u Exit keypad-shifted mode.

The console keyboard sends the expected ASCII characters for the usual alphabetic, numeric, and punctuation keys. The numeric keypad normally sends editing escape sequences, as described below. When shifted or in num-lock mode, it sends '0' to '9' and '.' instead. In num-lock mode (i.e., when the <num-lock> key is depressed, **<shift>** restores the normal escape sequences. In alternate-keypad mode, the numeric keypad sends "**<esc>? p**" to "**<esc>? y**" for '0' to '9' and "**<esc>? n**" for '.'.

<home> Send "cursor home" (**<esc>[H**).

<up> Send "cursor up" (**<esc>[A**).

<pg up> Send (**<esc>[V**).

<left> Send "cursor left" (**<esc>[D**).

<right> Send "cursor right" (**<esc>[C**).

<end> Send cursor to bottom left of screen (**<esc>[24 H**).

<down> Send "cursor down" (**<esc>[B**).

<pg dn> Move cursor to previous page (**<esc>[U**).

<ins> Toggle insert mode (**<esc>[@**).

<del> Delete the character at the cursor (**<esc>[P**).

The effects of the remaining keys are described below:

F1-F10 Send **<esc>[1 x ... <esc>[9 x, <esc>[0 x**.

<alt>F1-F10 Send **<esc>[1 y ... <esc>[9 y, <esc>[0 y**.

<esc> Mark the beginning of an escape sequence; **<esc><esc>** sends ASCII **ESC.**

<tab> Send ASCII **HT**.

<ctrl> When combined with 'A' through '_', send the corresponding ASCII control character; when combined with **<return>**, send ASCII **LF**; when combined with **<backspace>** send ASCII **DEL**; when combined with **<alt>** and **<del>**, issue system reset. **<ctrl-X>** cancels an escape sequence.

<shift> Change alphabetic keys from lower case to upper case, or from upper case to lower case in "caps lock" mode.

<alt> When combined with **<ctrl-alt-del>**, issue a system reset.

<backspace> Send ASCII **BS**; when combined with **<ctrl>**, send ASCII **DEL**.

<return> Send ASCII **CR**; when combined with **<ctrl>**, send ASCII **LF**.

***** Send ASCII '*'.

<caps lock> Toggle "caps lock" mode.

<num lock> Toggle the interpretation of the numeric keypad, as described above.

<scroll lock> Toggle console output, like **<ctrl-S>** and **<ctrl-Q>**.

- Send '-'.

+ Send '+'.

Files

/dev/console

See Also

ASCII, device drivers, signal()

const — C Keyword

Qualify an identifier as not modifiable

The type qualifier **const** marks an object as being unmodifiable. An object declared as being **const** cannot be used on the left side of an assignment (an *lvalue*), or have its value modified in any way. Because of these restrictions, an implementation may place objects declared to be **const** into a read-only region of storage.

See Also

C keywords, volatile

Notes

Mark Williams C recognizes this keyword, but its semantics are not yet implemented. Thus, storage declared with the **const** qualifier will *not* be treated as unmodifiable by the compiler, and no warnings will be generated.

const.h — Header File

Declare machine-dependent constants
#include <sys/const.h>

The header file **const.h** declares most machine-dependent constants. These are constants that change among the various machines for which the COHERENT system is available; an example is the clock speed of the processor.

See Also

header files, times()

continue — C Keyword

Force next iteration of a loop

continue forces the next iteration of a **for**, **while**, or **do** loop. For example,

```
while ((foo = getchar()) != EOF) {
     if ((foo < 'a') || (foo > 'z'))
          continue;
     ...        /* do something */
}
```

forces the **while** loop to throw away everything except lower-case alphabetic characters.

See Also

C keywords, for, while

continue — Command

Terminate current iteration of shell construct
continue [*n*]

The command **continue** helps to control the flow of commands given to the shell. When it is used without an argument, **continue** terminates the execution of the current iteration of the innermost **for**, **until**, or **while** shell construct; that is, it acts like a branch to the enclosing **done**, after which loop execution may continue or terminate. If an argument is given, **continue** terminates the current iteration of the *n*th enclosing **for**, **until**, or **while** loop.

The shell executes **continue** directly.

See Also

break, commands, for, ksh, sh, until, while

conv — Command

Numeric base converter
conv [*number*]

conv converts *number* to hexadecimal, decimal, octal, binary, and ASCII characters, and prints the results on the standard output. If no *number* is given, **conv** reads one number per line from the standard input until you type the end-of-file character **<ctrl-D>**.

number may be in hexadecimal, decimal, octal, binary, or character format, as shown below. Each example represents the decimal number 97.

Base	*Representation*
hexadecimal	**0x61**
hexadecimal	**#61**
decimal	**97**
octal	**0141**
binary	**$1100001**
character	**'a'**

conv represents an ASCII control character in its output by preceding the character by a carat '^'. For example, it prints **<ctrl-X>** as '^X'. **conv** prints "bad digit" if anything is wrong with the input.

See Also

bc, commands, conv, dd, od, units

Notes

conv represents the input *number* internally as a **long** integer. If *number* does not fit in a **long**, **conv** silently truncates it.

core — File Format

Core dump file format
#include <sys/uproc.h>

When a process terminates abnormally because of a process fault or because it receives an asynchronous signal from another process, COHERENT tries to write a memory dump of the process into a file called **core.** This file contains an image of the process code, data segments, the system description segment for the aborted process. The following lists the segment types and the symbolic names of their locations in the file:

SIUSERP	User process description segment
SISTACK	User stack segment
SISTEXT	Shared text segment
SIPTEXT	Private text segment
SISDATA	Shared data segment
SIPDATA	Private data segment

Not every dump necessarily contains all of the above segments. Neither shared text nor shared data segments are dumped. They are write-protected in memory, and the load module that was running when the dump occurred contains shared segment data.

The best way for a program (such as a debugger) to read the **core** file is to first read the user process description segment, which is always at the front and has a fixed size. It should be read into an area **UPASIZE** bytes long, but referenced with structured type **UPROC** (somewhat smaller than **UPASIZE** because of the system stack, which contains the user registers and other information in fixed places).

The **u_segl** member of the **UPROC** structure is a list of segment reference descriptors that contain the virtual address and length of each segment, which correspond exactly to its size in the dump. **NUSEG** segments are possible; the flag **SRFDUMP** in the field **sr_flag** indicates that a segment was dumped. By using the above method, you can use the entire file to reference program data and code at the time of the dump.

Other information found in the user process structure may be pertinent; the header file **sys/uproc.h** contains more information.

See Also

db, file formats, kill, l.out.h, signal(), wait()

Diagnostics

COHERENT will not write **core** if it already exists as a non-ordinary file or if there is more than one link to it. The 0200 bit in the status returned to the parent process by **wait** indicates a successful dump.

cos() — Mathematics Function (libm)

Calculate cosine
#include <math.h>
double cos(*radian***) double** *radian***;**

cos calculates the cosine of its argument *radian*, which must be in radian measure.

Example

For an example of this function, see the entry for **acos**.

See Also

mathematics library

cosh() — Mathematics Function (libm)

Calculate hyperbolic cosine
#include <math.h>
double cosh(*radian***) double** *radian***;**

cosh calculates the hyperbolic cosine of *radian*, which is in radian measure.

Example

The following program prompts you for a number; it then uses **cosh**, **sinh**, and **tanh** to generate, respectively, the hyperbolic cosine, sine, and tangent of your number.

```
#include <math.h>
#include <stdio.h>
#define display(x) dodisplay((double)(x), #x)

dodisplay(value, name)
double value; char *name;
{
     if (errno)
          perror(name);
     else
          printf("%10g %s\n", value, name);
     errno = 0;
}

main()
{
     extern char *gets();
     double x;
     char string[64];

     for(;;) {
          printf("Enter number: ");
          if(gets(string) == NULL)
               break;
          x = atof(string);

          display(x);
          display(cosh(x));
          display(sinh(x));
          display(tanh(x));
     }
}
```

See Also

mathematics library

Diagnostics

When overflow occurs, **cosh** returns a huge value that has the same sign as the actual result.

cp — Command

Copy a file

cp [-d] *oldname newname*
cp [-d] *file1 ... fileN directory*

cp copies files. In its first form, **cp** copies the contents of *oldname* to *newname*, which it creates if necessary. If *newname* is a directory, **cp** copies *oldname* to a file of the same name in directory *newfile*.

In its second form, **cp** copies each *file*, from *file1* through *fileN*, into *directory*.

With the **-d** option, **cp** preserves the date (modification time) of the source file or files on the target file or files. By default, target files get the current time.

A file cannot be copied to itself.

See Also

commands, ksh, mv, sh, wildcards

cpdir — Command

Copy directory hierarchy
cpdir [*option* ...] *dir1 dir2*

cpdir copies source directory hierarchy *dir1* to target hierarchy *dir2*, which is created if necessary. Either hierarchy may straddle device boundaries.

cpdir preserves as much as possible of the source structure. Files under *dir1* go to identically named files under *dir2*. Links between source files are preserved as links between corresponding target files. Preserved source file attributes include mode, subject to the user's file creation mask. If the user is not the superuser, **cpdir** cannot preserve the owner, group, and sticky bits in the mode, and the invoking user owns all new files; under the superuser it preserves these as well. In addition, the superuser may "copy" special nodes and pipe nodes; **cpdir** copies only the facility, not the contents. It also preserves real major and minor device numbers of special nodes.

If the target file corresponding to a source file exists and is not a directory, **cpdir** unlinks it before copying. This differs from the action of **cp**.

cpdir recognizes the following options:

-a Give a verbose account on one line of the files copied.

-d Preserve the last-modified date instead of using the present date.

-e Print error message and continue execution after an error. The default action is to exit on any error.

-r [*n*] Descend no more than n levels in the source hierarchy. Contents of *dir1* are at level 1. If missing, *n* defaults to 1.

-s *name*
Suppress the copy of file *name*, which should be the pathname of the file relative to *dir1*.

-t Test only, make no changes. With this option, **cpdir** prints a report of all errors (**-e** is implied), all unlinked target files, and other useful information, including a summary of all external links into the target hierarchy that would have been broken had the unlinking actions been executed.

-u Update regular files. Copy the source only if it was created or altered more recently than the target file, or if the target does not exist.

-v Print a verbose account of its activities. **cp** prints a file-by-file account of its actions, in addition to the information listed under **-t**.

See Also

cp, commands, link(), umask(), unlink()

cpio — Command

Archiving/backup utility
cpio -o[Bacv]
cpio -i[Bcdfmrtuv] [*pattern...*]
cpio -p[adlmruv] *directory*

cpio is an archiving utility that reads and writes files in the format specified by the **cpio Archive/Interchange File Format** specified in IEEE standard 1003.1-1988.

Options

cpio recognizes the following command-line options:

-a Reset the access times of input files after they have been copied. When the **-l** option is also specified, the linked files do not have their access times reset. Can be used only with the **-o** or **-i** options.

-B Change the size of a block. Input/output is to be blocked 5,120 bytes to the record. This option can be used only with the **-o** or **-i** options, for data directed to or from character-special files.

-c Write header information in ASCII characters for portability. Can be used only with the **-i** or **-o** options.

-d Creates directories as needed. Can be used only with the **-i** or **-p** options.

-f Copy all files except those in whose names match a *pattern*. Can be used only with the **-i** option.

-i In. Read the standard input, which it assumes to be an archive that had been created with the **-o** option to **cpio**. Copy all files within the archive whose names match a *pattern* into the current directory (default, all files).

-l Whenever possible, link files rather than copying them. Can be used only with the **-p** option.

-m Retain previous modification times. This option is ineffective on directories that are being copied. Can be used only with the **-i** or **-p** options.

-o Out. Copy all files whose names match a *pattern* (default, all files) into an archive written to standard output.

-p Pass mode. This option causes **cpio** to read standard input for a list of file names to copy to destination *directory*. This mode of operation is similar in functionality to command **cpdir**, with the added ability to specify individual file names via standard input.

-r Interactively rename files. Before it copies a file, **cpio** asks you to rename the file. If you type just **<return>**, **cpio** skips the file. Should be used only with the **-i** or **-o** options.

-t Print a table of contents of an existing archive; do not copy files from the archive. Can be used only with the **-i** option.

-u Copy files unconditionally. Usually an older file will not replace a new file with the same name. Can be used only with the **-i** or **-p** options.

-v Verbose option: print the names of all affected files. Can be used only with the **-i** option. Provides a detailed listing when used with the **-t** option.

Operands

The following operands are available:

pattern
: This names the files to be manipulated by **cpio**. This can be a simple regular expression.

directory
: The destination directory.

Multi-Volume Operation

cpio supports multi-volume backups and restoration when using the "raw" (i.e., character-special) floppy device. If a **cpio** archive exceeds the size of one floppy diskette, **cpio** will issue a prompt of the form:

```
Ready for volume 2
Type "device/name" when ready to proceed ...
```

Remove the first diskette and insert the next logical diskette. Enter the name of the floppy device, e.g. **/dev/rfha0** or **/dev/rfva1**, and press **<Enter>**. Please note well that multi-volume operation requires the raw floppy device and the use of pre-formatted floppy diskettes.

Examples

The following command copies all files and directories listed by the command **find** and copies them into the archive **newfile.cpio**:

```
find . -print | cpio -oc > ../newfile.cpio
```

The following command reads the **cpio** archive **newfile.cpio** and extracts all files whose names match the patterns **memo/al** or **memo/b***:

```
cpio -icdv "memo/al" "memo/b*" <../newfile.cpio
```

Note that the **-d** option forces **cpio** to create the sub-directory **memo** and write the files into it. Otherwise, the files would have been written into the current directory. Option **-v** causes **cpio** to display each file name as it is extracted from the archive.

The following commands perform a multi-volume backup of all files on mounted filesystem **/v** to the character-special (i.e., "raw") floppy device **/dev/rfha0**:

```
su root
cd /v
find . -print | cpio -ocv >/dev/rfha0
```

If the **cpio** archive exceeds one diskette, you will be prompted to insert another formatted diskette.

See Also

commands, dump, pax, tar, ustar

Notes

cpio has the following restrictions:

- Path names are restricted to 256 characters.
- You must have appropriate privileges to copy special files.

- Blocks are reported in 512-byte quantities.

cpio was developed by Mark H. Colburn and sponsored by The USENIX Association. Copyright © 1989 by Mark H. Colburn (mark@jhereg.MN.ORG). All rights reserved. See the compressed **tar** archive **/usr/src/alien/pax.tar.Z** for full descriptions of copyright, restrictions, and licensing terms.

cpio is provided in binary form per the licensing terms set forth by the author. It is distributed as a service to COHERENT customers, as is. It is not supported by Mark Williams Company. *Caveat utilitor.*

cpp — Command

C preprocessor
/lib/cpp [*option*...] [*file*...]

The command **cpp** calls the C preprocessor to perform C preprocessing. It performs the operations described in section 3.8 of the ANSI Standard; these include file inclusion, conditional code selection, constant definition, and macro definition. See the entry on **preprocessing** for a full description of the C's preprocessing language.

Normally, **cpp** is used to preprocess C programs, but it can be used as a simple macro processor for other types of files as well. **cpp** reads each input *file*, processes directives, and writes its product on **stdout**. If the option **-E** is not used, **cpp** also writes into its output statements of the form **#line***n filename*, so that the parser will be able to connect its error messages and debugger output with the original line numbers in your source files.

Options

The following summarizes **cpp**'s options:

-D*VARIABLE*
: Define *VARIABLE* for the preprocessor at compilation time. For example, the command

```
cc -DLIMIT=20 foo.c
```

tells the preprocessor to define the variable **LIMIT** to be 20. The compiled program acts as though the directive **#define LIMIT 20** were included before its first line.

-E
: Strip all comments and line numbers from the source code. This option is used to preprocess assembly-language files or other sources, and should not be used with the other compiler phases.

-I *directory*
: C allows two types of **#include** directives in a C program, i.e., **#include "file.h"** and **#include <file.h>**. The **-I** option tells **cpp** to search a specific directory for the files you have named in your **#include** directives, in addition to the directories that it searches by default. You can have more than one **-I** option on your **cc** command line.

-o *file*
: Write output into *file*. If this option is missing, **cpp** writes its output onto **stdout**, which may be redirected.

-U*VARIABLE*
: Undefine *VARIABLE*, as if an **#undef** directive were included in the source program. This is used to undefine the variables that **cpp** defines by default.

See Also

C preprocessor, cc, commands

creat() — COHERENT System Call (libc)

Create/truncate a file
int creat(*file***, *** mode***) char ****file***; **int** *mode***;**

creat creates a new *file* or truncates an existing *file*. It returns a file descriptor that identifies *file* for subsequent system calls. If *file* already exists, its contents are erased. In this case, **creat** ignores the specified *mode*; the mode of the *file* remains unchanged. If *file* did not exist previously, **creat** uses the *mode* argument to determine the mode of the new *file*. For a full definition of file modes, see **chmod** or the header file **stat.h**. **creat** masks the *mode* argument with the current **umask**, so it is common practice to create files with the maximal mode desirable.

Example

For an example of how to use this routine, see the entry for **open**.

See Also

chmod(), COHERENT system calls(), fopen(), open(), stat.h, STDIO

Diagnostics

If the call is successful, **creat** returns a file descriptor. It returns -1 if it could not create the file, typically because of insufficient system resources or protection violations.

cron — System Maintenance

Execute commands periodically
/etc/cron&

cron is a daemon that executes commands at preset times. The commands and their scheduled execution times are kept in the file **/usr/lib/crontab**.

Once each minute **cron** searches through **crontab**. For each command stored there, **cron** compares the current time with the scheduled execution time and executes the command if the times match. When it finishes the search, **cron** sleeps until the next minute. Because it never exits, **cron** should be executed only once (customarily by **/etc/rc**).

crontab consists of lines separated by newlines. Each line consists of six fields separated by white space (tabs or blanks). The first five fields describe the scheduled execution time of the command. Respectively, they represent the minute (0-59), hour (0-23), day of the month (1-31), month of the year (1-12), and day of the week (0-6, 0 indicates Sunday). Each field may contain a single integer in the appropriate range, a pair of integers separated by a hyphen '-' (meaning all integers between the two, inclusive), an asterisk '*' (meaning all legal values), or a comma-separated list of the above forms. The remainder of the line gives the command to be executed at the given time.

For example, the **crontab** entry

```
29 * * 7 0 msg henry Succotash!
```

means that every hour on the half-hour during each Sunday in July, **cron** will invoke the command **msg**, and the user named **henry** will have the message

```
daemon: Succotash!
```

written on his terminal's screen (if he is logged in).

cron recognizes three special characters and escape sequences in the **crontab**. If a

command contains the percent character '%', **cron** executes only the portion up to the first '%' as a command and passes the remainder to the command as its standard input. **cron** translates any percent characters '%' in the remainder to newlines. The special interpretation of '%' can be prevented by preceding it with a backslash, '\%'. Finally, **cron** removes the sequence **<newline>** from the text before passing it to the shell **sh**; this can be used to make an entry in the **crontab** more legible.

cron is designed for commands that must be executed regularly. Temporal commands that need to be executed only once should be handled with the command **at**.

Permissions

cron performs some interesting manipulations with permissions. This is necessary to allow **cron** to run a wide variety of programs untended without creating loopholes in the system's security. Occasionally, this can create difficulties for users who do not grasp what **cron** does or why. The following describes how **cron** manipulates permissions on the programs you ask it to run.

To begin, when **cron** runs an entry from **/usr/lib/crontab** with user ID and group ID of user **daemon**. This prevents security holes involving entries in **/etc/crontab**. For example, the following **crontab** entry contains redirection:

```
* * * * *  echo hello world >/dev/console 2>&1
```

cron will not execute it if user **daemon** lacks permission to write to **/dev/console**.

To clarify further,

```
* * * * *  echo hello world >/dev/console 2>/tmp/cron.err
```

does not even create **/tmp/cron.err** if **daemon** cannot write to **/dev/console**, whereas

```
* * * * *  echo hello world >/tmp/cron.log 2>/tmp/cron.err
```

works as expected if **daemon** can write into **/tmp**).

When the shell executes a command in the background, it reads its standard input from **/dev/null** (unless redirected) and writes its standard output to the controlling tty. If **cron** is invoked with **/etc/cron&** from **/etc/rc**, there is no controlling tty, so the standard output goes to **/dev/null**. Thus,

```
* * * * *  echo hello world
```

typically writes **hello world** to **/dev/null**. If the superuser kills the **/etc/cron** invoked from **/etc/rc** and reinvokes **/etc/cron&** directly from the console, this command echoes **hello world** to the console every minute.

When a user logs in, **/bin/login** grabs the tty and runs **chown** and **chmod** on it. It is owned by the user with default permissions 700. If the user logged in on the console says

```
chmod /dev/console a+w
```

to allow all users to write to it, then the **crontab** entry

```
* * * * *  echo hello world >/dev/console 2>/tmp/cron.err
```

would indeed echo to the console every minute.

Files

/usr/lib/crontab for stored commands

See Also

at, init, system maintenance

crypt() — General Function (libc)

Encryption using rotor algorithm
char *crypt(*key, extra***); char ****key***, ****extra***;**

crypt implements a version of rotor encryption. **crypt** produces encrypted passwords that are verified by comparing the encrypted clear text against an original encryption.

key is an ASCII string that contains the user's password. *extra* is a string of two additional characters, stored in the password file with the encrypted password. Each character must come from an alphabet of 64 symbols, which consists of the upper-case and lower-case letters, digits, the period '.', and the slash '/'.

crypt returns a string built from the 64-character alphabet described above; the first two characters returned are the *extra* argument, and the rest contain the encrypted password.

See Also

ASCII, general functions

crypt — Command

Encrypt/decrypt text
crypt [*password*]

The command **crypt** encrypts data. It emulates a rotor-encryption machine, such as the Enigma or Hagelin C-48 cipher machines. Unlike these machines, **crypt** uses only one rotor, with a 256-character alphabet and a keying sequence of period 2^32.

crypt reads text from standard input and writes the encrypted text to standard output. *password* is used to construct the model of the machine and to start the keying sequence. If no *password* is given, **crypt** prompts for a password on the terminal and disables echo while it is being typed in. The *password* may be up to ten characters long, but must not be empty; all characters past the first ten are ignored. All characters are legal, although it may not be possible to input certain characters from the terminal.

crypt uses the same *password* for both encryption and decryption. For example, the commands

```
crypt COHERENT <file1 >file2
crypt COHERENT <file2 >file3
```

leave *file3* identical to *file1*.

Encrypted files produced by **ed** with its **-x** option may be read by **crypt**, and vice versa, as **ed** uses **crypt** to perform its encryption.

Security of a cryptosystem depends on several factors:

1. Brute-force attempts to break the system should be infeasible. Passwords should be at least five characters long; although the construction of the machine model from the password takes a substantial fraction of a second, it is still plausible that encrypted files could be read by a brute-force search of a portion of the password space (say, all passwords less than four characters long).
2. Cryptanalysis of the basic encryption scheme should be very hard. Analysis of rotor machines is understood, but it is difficult and in most cases probably not worth the trouble.

3. Passwords must be kept secret. **crypt** erases *password* as soon as it can, to avoid the possibility that it could appear in the output of **ps**.

4. Privileged access to the system must be guarded. Under COHERENT, the security of **crypt** can be no better than the security governing access to superuser status, because the superuser can do practically anything. This is probably **crypt**'s most vulnerable point.

Files

/dev/tty — Typed passwords

See Also

commands

Kahn D: *The Code Breakers*. New York, Macmillan, 1967.

Morris R: The Hagelin cipher machine (M-209). *Cryptologia*, July 1978.

ct — Device Driver

Controlling terminal driver

For each process, the controlling terminal driver **/dev/tty** is an interface to the appropriate terminal driver. COHERENT passes any input-output call (e.g. **close**, **ioctl**, **open**, **read**, or **write**) on this special file directly to the controlling terminal device for the requesting process.

Normally, the controlling terminal is the default standard input, output, and error device. This is not the case for daemon processes started by the initial process.

Files

/dev/tty

See Alsc

device drivers, init

Diagnostics

When a call finds no valid controlling terminal for a process, it returns a value of -1 and sets **errno** to **ENXIO.**

ctags — Command

Generate tags and refs files for vi editor

ctags [-r] *files*...

ctags generates the files **tags** and **refs** from a group of C-source files. **tags** is used by the **elvis** editor's **:tag** command, **<ctrl-]>** command, and **-t** option. **refs** is used by the command **ref**.

Each C-source file is scanned for **#define** statements and global function definitions. The name of the macro or function becomes the name of a tag. For each tag, a line is added to **tags**, which contains the following:

- the name of the tag
- a tab character
- the name of the file containing the tag
- a tab character
- a way to find the particular line within the file

refs is used by the command **ref**, which can be invoked via **elvis**'s **K** command. When **ctags** finds a global function definition, it copies the function header into **refs**. The first line

is flush against the right margin, but the argument definitions are indented. The command **ref** can search **refs** much faster than it could search all C-source files. The file-names list will typically include the names of all C-source files in the current directory, in the following format:

```
ctags -r *.[ch]
```

The **-r** to **ctags** tells it to generate both **tags** and **refs**. Without **-r**, it generates only **tags**.

See Also

commands, elvis, ref

Notes

This version of **ctags** does not parse ANSI source code very well. It has trouble recognizing the ANSI function definitions.

ctags is copyright © 1990 by Steve Kirkendall, and was written by Steve Kirkendall (kirkenda@cs.pdx.edu or ...uunet!tektronix!psueea!eecs!kirkenda), assisted by numerous volunteers. It is freely redistributable, subject to the restrictions noted in included documentation. Source code for **ctags** is available through the Mark Williams bulletin board, USENET, and numerous other outlets.

Please note that this program is offered as a service to COHERENT users, but is not supported by Mark Williams Company. *Caveat utilitor.*

ctime() — Time Function (libc)

Convert system time to an ASCII string
#include <time.h>
#include <sys/types.h>
char *ctime(*timep***) time_t *****timep***;

ctime converts the system's internal time into a string that can be read by humans. It takes a pointer to the internal time type **time_t**, which is defined in the header file **time.h**, and returns a fixed-length string of the form:

```
Thu Mar  7 11:12:14 1989\n
```

time_t is defined in the header **types.h**.

ctime is implemented as a call to **localtime** followed by a call to **asctime**.

Example

For another example of this function, see the entry for **asctime**.

```
#include <time.h>
#include <sys/types.h>

main()
{
	time_t t;

	time(&t);
	printf(ctime(&t));
}
```

See Also

time, time.h

Notes

ctime returns a pointer to a statically allocated data area that is overwritten by successive calls.

ctype — Overview

#include <ctype.h>

The **ctype** macros and functions test a character's *type*, and can transform some characters into others. They are as follows:

isalnum()	Test if alphanumeric character
isalpha()	Test if alphabetic character
isascii()	Test if ASCII character
iscntrl()	Test if a control character
isdigit()	Test if a numeric digit
islower()	Test if lower-case character
isprint()	Test if printable character
ispunct()	Test if punctuation mark
isspace()	Test if a tab, space, or return
isupper()	Test if upper-case character
_tolower()	Change to lower-case character
_toupper()	Change to upper-case character

These are defined in the header file **ctype.h**, and each is described further in its own Lexicon entry.

Example

The following example demonstrates the macros **isalnum**, **isalpha**, **isascii**, **iscntrl**, **isdigit**, **islower**, **isprint**, **ispunct**, and **isspace**. It prints information about the type of characters it contains.

```
#include <ctype.h>
#include <stdio.h>

main()
{
     FILE *fp;
     char fname[20];
     int ch;
     int alnum = 0;
     int alpha = 0;
     int allow = 0;
     int control = 0;
     int printable = 0;
     int punctuation = 0;
     int space = 0;

     printf("Enter name of text file to examine: ");
     fflush(stdout);
     gets(fname);
```

```
        if ((fp = fopen(fname, "r")) != NULL) {
             while ((ch = fgetc(fp)) != EOF) {

                  if (isascii(ch)) {
                       if (isalnum(ch))
                            alnum++;
                       if (isalpha(ch))
                            alpha++;
                       if (islower(ch))
                            allow++;
                       if (iscntrl(ch))
                            control++;
                       if (isprint(ch))
                            printable++;
                       if (ispunct(ch))
                            punctuation++;
                       if (isspace(ch))
                                 space++;

                  } else {
                       printf("%s is not ASCII.\n",
                                 fname);
                       exit(1);
                  }
             }

             printf("%s has the following:\n", fname);
             printf("%d alphanumeric characters\n", alnum);
             printf("%d alphabetic characters\n", alpha);
             printf("%d alphabetic lower-case characters\n",
                  allow);
             printf("%d control characters\n", control);
             printf("%d printable characters\n", printable);
             printf("%d punctuation marks\n", punctuation);
             printf("%d white space characters\n", space);
             exit(0);

        } else  {
             printf("Cannot open \"%s\".\n", fname);
             exit(2);
        }
   }
```

See Also

ctype.h, libraries

ctype.h — Header File

Header file for data tests
#include <ctype.h>

ctype.h is a header file that holds the texts of the macros described in the overview entry **ctype**.

See Also

ctype, header files

curses — Overview

Library of screen-handling functions

curses is a set of routines that allow you to manipulate the screen in a sophisticated manner. These routines use the **termcap** functions to read information about the user's terminal. This allows you to write programs that can perform rudimentary graphics on a wide variety of terminals.

curses contains routines that do the following:

- Move the cursor about the screen.
- Insert text onto the screen, either in normal or reverse video (if supported by the display device).
- Read what is typed by the user and display it properly.
- Organize the screen into one or more rectangular regions, or *windows*, optionally draw a border around each, and manage each independently.

curses organizes the screen into a two-dimensional array of cells, one cell for every character that the device can display. It maintains in memory an image of the screen, called the *curscr*. A second image, called the *stdcur*, is manipulated by the user; when the user has finished a given manipulation, **curses** copies the changes from the *stdcur* to the *curscr*, which results in their being displayed on the physical screen. This act of copying from the *stdscr* to the *curscr* is called *refreshing* the screen. **curses** keeps track of where all changes have begun and ended between one refresh and the next; this lets it rewrite only the portions of the *curscr* that the user has changed, and so speed up rewriting of the screen.

curses records the position of a "logical cursor", which points to the position in the *stdscr* that is being manipulated by the user, and also records the position of the physical cursor. Note that the two are not necessarily identical: it is possible to manipulate the logical cursor without repositioning the physical cursor, and vice versa, depending on the task you wish to perform.

Most **curses** routines work by manipulating **WINDOW** object. **WINDOW** is defined in the header **curses.h** as follows:

```
#define WINDOW _win_st
struct _win_st {
        short           _cury, _curx;
        short           _maxy, _maxx;
        short           _begy, _begx;
        short           _flags;
        short           _ch_off;
        bool            _clear;
        bool            _leave;
        bool            _scroll;
        char            **_y;
        short           *_firstch;
        short           *_lastch;
        struct _win_st  *_nextp, *_orig;
};
```

Type **bool** is defined in **curses.h**; an object of this type can hold the value of true (nonzero) or false (zero).

The following describes each **WINDOW** field in detail.

_cury, _curx Give the Y and X positions of the logical cursor. The upper left corner of the window is, by definition, position 0,0. Note that **curses** by convention gives positions as Y/X (column/row) rather than X/Y, as is usual elsewhere.

_maxy, _maxx Width and height of the window.

_begy, _begx Position of the upper left corner of the window relative to the upper left corner of the physical screen. For example, if the window's upper left corner is five rows from the top of the screen and ten columns from the left, then **_begy** and **_begx** will be set to ten and five, respectively.

_flags One or more of the following flags, logically OR'd together:

_SUBWIN — Window is a sub-window
_ENDLINE — Right edge of window touches edge of the screen
_FULLWIN — Window fills the physical screen
_SCROLLWIN — Window touches lower right corner of physical screen
_FULLINE — Window extends across entire physical screen
_STANDOUT — Write text in reverse video
_INSL — Line has been inserted into window
_DELL — Line has been deleted from window

_ch_off Character offset.

_clear Clear the physical screen before next refresh of the screen.

_leave Do not move the physical cursor after refreshing the screen.

_scroll Enable scrolling for this window.

_y Pointer to an array of pointers to the character arrays that hold the window's text.

_firstch Pointer to an array of integers, one for each line in the window, whose value is the first character in the line to have been altered by the user. If a line has not been changed, then its corresponding entry in the array is set to **_NOCHANGE**.

_lastch Same as **_firstch**, except that it indicates the last character to have been changed on the line.

_nextp Point to next window.

_orig Point to parent window.

When **curses** is first invoked, it defines the entire screen as being one large window. The programmer has the choice of subdividing an existing window or creating new windows; when a window is subdivided, it shares the same *curscr* as its parent window, whereas a new window has its own *stdscr*.

Mark Williams Company will document its **curses** library in full in a later release of this manual. The following table, however, summarizes the functions and macros that that compose the **curses** library.

addch(*ch*) **char** *ch*;
Insert a character into *stdscr*.

addstr(*str*) **char** *****str*;
Insert a string into *stdscr*.

box(*win, vert, hor*) **WINDOW** *****win*; **char** *vert, hor*;
Draw a box. *vert* is the character used to draw the vertical lines, and *hor* is used to draw the horizontal lines. For example

```
box(win, '|', '-');
```

draws a box around window **win**, using '|' to draw the vertical lines and '-' to draw the horizontal lines.

clear() Clear the *stdscr*.

clearok(*win,bf*) **WINDOW** *****win*; **bool** *bf*;
Set the clear flag for window *win*. This will clear the screen at the next refresh, but not reset the window.

clrtobot()
Clear from the position of the logical cursor to the bottom of the window.

clrtoeol()
Clear from the logical cursor to the end of the line.

crmode()
Turn on control-character mode; i.e., force terminal to receive cooked input.

delch() Delete a character from *stdscr*; shift the rest of the characters on the line one position to the left.

deleteln()
Delete all of the current line; shift up the rest of the lines in the window.

delwin(*win*) **WINDOW** *****win*;
Delete window *win*.

echo() Turn on both physical and logical echoing; i.e., character are automatically inserted into the current window and onto the physical screen.

endwin()
Terminate text processing with **curses**.

erase() Erase a window; do not clear the screen.

getch() Read a character from the terminal.

getstr(*str*) **char** *****_str_**;**
Read a string from the terminal.

getyx(*win,y,x*) **WINDOW** *****_win_**; short** *y,x*;
Read the position of the logical cursor in *win* and store it in *y,x*. Note that this is a macro, and due to its construction the variables *y* and *x* must be integers, not pointers to integers.

inch() Read the character pointed to by the *stdscr*'s logical cursor.

WINDOW *initscr()
Initialize **curses**.

insch(*ch*) **char** *ch*;
Insert character *ch* into the *stdscr*.

insertln()
Insert a blank line into *stdscr*, above the current line.

leaveok(*win,bf*) **WINDOW** *****_win_**; bool** *bf*;
Set *win*->**_leave** to *bf*. If set to **TRUE**, refresh will leave the cursor after the last character changed by refresh. This makes sense if you want to minimize the commands sent to the screen and it does not matter where the cursor is.

char *longname(*termbuf, name*) **char** *****_termbuf_**,** *****_name_**;**
Copy the long name for the terminal from *termbuf* into *name*.

move(*y,x*) **short** *y,x*;
Move logical cursor to position *y,x* in *stdscr*.

mvaddbytes(*y,x,da,count*) **int** *y,x*; **char** *****_da_**; int** *count*;
Move to position *y,x* and print *count* bytes from the string pointed to by *da*.

mvaddch(*y,x,ch*) **short** *y,x*; **char** *ch*;
Move the logical cursor to position *y,x* and insert character *ch*.

mvaddstr(*y,x,str*) **short** *y,x*; **char** *****_str_**;**
Move the logical cursor to position *y,x* and insert string *str*.

mvcur(*y_cur,x_cur,y_new,x_new*) **int** *y_cur, x_cur, y_new, x_new*;
Move cursor from position *y_cur,x_cur* to position *y_new,x_new*.

mvdelch(*y,x*) **short** *y,x*;
Move to position *y,x* and delete the character found there.

mvgetch(*y,x*) **short** *y,x*;
Move to position *y,x* and get a character through *stdscr*.

mvgetstr(*y,x,str*) **short** *y,x*; **char** *****_str_**;**
Move to position *y,x*, get a string through *stdscr*, and copy it into *string*.

mvinch(*y,x*) **short** *y,x*;
Move to position *y,x* and get the character found there.

mvinsch(*y,x,ch*) **short** *y,x*; **char** *ch*;
Move to position *y,x* and insert a character into *stdscr*.

mvwaddbytes(*win,y,x,da,count*) **WINDOW** **win*; **int** *y,x*; **char** **da*; **int** *count*;
Move to position *y,x* and print *count* bytes from the string pointed to by *da* into window *win*.

mvwaddch(*win,y,x,ch*) **WINDOW** **win*; **int** *y,x*; **char** *ch*;
Move to position *y,x* and insert character *ch* into window *win*.

mvwaddstr(*win,y,x,str*) **WINDOW** **win*; **short** *y,x*; **char** **str*;
Move to position *y,x* and insert character *ch*.

mvwdelch(*win,y,x*) **WINDOW** **win*; **int** *y,x*;
Move to position *y,x* and delete character *ch* from window *win*.

mvwgetch(*win,y,x*) **WINDOW** **win*; **short** *y,x*;
Move to position *y,x* and get a character.

mvwgetstr(*win,y,x,str*) **WINDOW** **win*; **short** *y,x*; **char** **str*;
Move to position *y,x*, get a string, and write it into *str*.

mvwin(*win,y,x*) **WINDOW** **win*; **int** *y,x*;
Move window *win* to position *y,x*.

mvwinch(*win,y,x*) **WINDOW** **win*; **short** *y,x*;
Move to position *y,x* and get character found there.

mvwinsch(*win,y,x,ch*) **WINDOW** **win*; **short** *y,x*; **char** *ch*;
Move to position *y,x* and insert character *ch* there.

WINDOW *newwin(lines, cols, y1, x1) **int** *lines, cols, y1, x1*;
Create a new window. The new window is *lines* lines high, *cols* columns wide, with the upper-left corner at position *y1,x1*.

nl() Turn on newline mode; i.e., force terminal to output **<newline>** after **<linefeed>**.

nocrmode()
Turn off control-character mode; i.e., force terminal to accept raw input.

noecho()
Turn off echo mode.

nonl() Turn off newline mode.

noraw()
Turn off raw mode.

overlay(*win1,win2*) **WINDOW** **win1*, *win2*;
Copy all characters, except spaces, from their current positions in *win1* to identical positions in *win2*.

overwrite(*win1,win2*) **WINDOW** **win1*, *win2*;
Copy all characters, including spaces, from *win1* to their identical positions in *win2*.

printw(*format[,arg1,...argN]*) **char** **format*; [*data type*] *arg1,..argN*;
Print formatted text on the standard screen.

raw() Turn on raw mode; i.e., kernel does not process what is typed at the keyboard, but passes it directly to **curses**. In normal (or *cooked*) mode, the kernel intercepts and processes the control characters *<ctrl-C>*, *<ctrl-S>*, *<ctrl-Q>*, and **<ctrl-Y>**. See the entry for **stty** for more information.

refresh()
Copy the contents of *stdscr* to the physical screen.

resetty()
Reset the terminal flags to values stored by earlier call to **savetty**.

savetty()
Save the current terminal settings.

scanw(*format[,arg1,...argN]***) char** ***format*; [*data type*] *arg1,..argN*;
Read the standard input; translate what is read into the appropriate data type.

scroll(*win***) WINDOW** ***win*;
Scroll *win* up by one line.

scrollok(*win,bf***) WINDOW** ***win*; **bool** *bf*;
Permit or forbid scrolling of window *win*, depending upon whether *bf* is set to true or false.

standend()
Turn off standout mode.

standout()
Turn on standout mode for text. Usually, this means that text will be displayed in reverse video.

WINDOW *subwin(*win,lines,cols,y1,x1***) int** *win,lines,cols,y1,x1*;
Create a sub-window in window *win*. New sub-window is *lines* lines high, *cols* columns wide, and is fixed at position *y1,x1*. Note that the position is relative to the upper-left corner of the physical screen.

touchwin(*win***) WINDOW** ***win*;
Copy all characters in window *win* to the screen.

waddch(*win,ch***) WINDOW** ***win*; **char** *ch*;
Add character *ch* to window *win*.

waddstr(*win,str***) WINDOW** ***win*; **char** ***str*;
Add the string pointed to by *str* to window *win*.

wclear(*win***) WINDOW** ***win*;
Clear window *win*. Move cursor to position 0,0 and set the screen's clear flag.

wclrtobot(*win***) WINDOW** ***win*;
Clear window *win* from current position to the bottom.

wclrtoeol(*win***) WINDOW** ***win*;
Clear window *win* from the current position to the end of the line.

wdelch(*win***) WINDOW** ***win*;
Delete the character at the current position in window *win*; shift all remaining characters to the right of the current position one position left.

wdeleteln(*win***) WINDOW** ***win*;
Delete the current line and shift all lines below it one line up.

werase(*win*) **WINDOW** **win*;
Clear window *win*. Move the cursor to position 0,0 but do not set the screen's clear flag.

wgetch(*win*) **WINDOW** **win*;
Read one character from the standard input.

wgetstr(*win,str*) **WINDOW** **win*; **char** **str*;
Read a string from the standard input; write it in the area pointed to by *str*.

winch(*win*) **WINDOW** **win*;
Force the next call to **refresh()** to rewrite the entire screen.

winsch(*win,ch*) **WINDOW** **win*; **char** *ch*;
Insert character *ch* into window *win* at the current position. Shift all existing characters one position to the right.

winsertln(*win*) **WINDOW** **win*;
Insert a blank line into window *win* at the current position. Move all lines down by one position.

wmove(*win,y,x*) **WINDOW** **win*; **int** *y*, *x*;
Move current position in the window *win* to position *y,x*.

wprintw(*win,format[,arg1,...argN]*) **WINDOW** **win*; **char** **format*; [*data type*] **arg1,..argN;**
Format text and print it to the current position in window *win*.

wrefresh(*win*) **WINDOW** **win*;
Refresh a window.

wscanw(*win,format[,arg1,...argN]*) **WINDOW** **win*; **char** **format*; [*data type*] *arg1,..argN*;
Read standard input from the current position in window *win*, format it, and store it in the indicated places.

wstandend(*win*) **WINDOW** **win*;
Turn off standout (reverse video) mode for window *win*.

wstandout(*win*) **WINDOW** **win*;
Turn on standout (reverse video) mode for window *win*.

These routines are declared and defined in the header file **curses.h**.

Structure of a curses Program

To use the **curses** routines, a program must include the header file **curses.h**, which declares and defines the functions and macros that comprise the **curses** library.

Before a program can perform any screen operations, it must call the function **initscr()** to initialize the **curses** environment.

As noted above, **curses** manipulates text in a copy of the screen that it maintains in memory. After a program has manipulated text, it must call **refresh()** to copy these alterations from memory to the physical screen. (This is done because writing to the screen is slow; this scheme permits mass alterations to be made to copy in memory, then written to the screen in a batch.)

Finally, when the program has finished working with **curses**, it must call the function **endwin()**. This frees memory allocated by **curses**, and generally closes down the **curses** environment gracefully.

Example

The following program, called **curexample.c**, gives a simple example of programming with **curses**. To compile this program, use the command line:

```
cc curexample.c -lcurses -lterm
```

Note that order in which the libraries are called is significant.

When this program is run, it clears the screen, then waits for you to type a Y coordinate, a space, and then an X coordinate. Note that these do not echo on the screen. It moves the cursor to the requested coordinates, and there display any non-numeric string that you type. If you type numerals, **curexample** will assume that you wish to move the cursor to a new location. To exit, type **<ctrl-C>**.

```
#include <ascii.h>
#include <ctype.h>
#include <curses.h>

#define NORMAL 0
#define INY  1
#define INX  2

main()
{
     int c, y, x, state;

     initscr(); /* initialize curses */
     noecho();
     raw();

     clear();
     move(0, 0);

     for(state = NORMAL;;) {
          refresh();
          c = getch();
          if(isdigit(c)) {

               switch (state) {
               case NORMAL:
                    y = x = 0;
                    state = INY;
               case INY:
                    y *= 10;
                    y += c - '0';
                    break;
               case INX:
                    x *= 10;
                    x += c - '0';
               }
          } else {
```

```
                if (c == A_ETX) { /* ctl-c */
                        noraw();
                        echo();
                        endwin();
                        exit(0);
                }

                switch (state) {
                case INX:
                        state = NORMAL;
                        move(y, x);
                case NORMAL:
                        addch(c);
                        break;
                case INY:
                        state = INX;
                }
        }
    }
}
```

See Also

curses.h, libraries, termcap Strang J: *Programming with curses.* Sebastopol, Calif, O'Reilly & Associates Inc., 1986.

Notes

curses is copyrighted © by the Regents of the University of California.

Please note that **curses** will not take any characters with the high bit set. **curses** is designed to allow communication with arbitrary terminals, most of which do not have these characters; therefore, this is a feature, not a bug.

curses.h — Header File

Define functions and macros in curses library
#include <curses.h>

curses.h defines the macros and declares the functions that comprise the **curses** library.

See Also

header files

cut — Command

Select portions of each line of its input
cut -c*list* [*file* ...]
cut -f*list* [**-s**] [**-d** *char*] [*file* ...]

cut selects portions of each line of its input and writes them to the standard output. *list* specifies the portions to select. **cut** reads its input from *file*, or the standard input by default.

list is a comma-separated set of numbers or number ranges. Number ranges consist of a number, a hyphen ('-'), and a second number, and select the fields or columns from the first number to the second, inclusive. Preceding a number or number range by a hyphen selects all fields or columns from one to the first number. Following a number or number range by

a hyphen selects all fields or columns from the last number to the end of the line. Numbers and number ranges may be repeated, overlap, and appear in any order. It is not an error to select a field or column not present in the input line.

cut recognizes the following command-line options:

-c*list* *list* specifies character positions.

-f*list* *list* specifies fields, delimited in the input by one **<tab>** character. Output fields are separated by one **<tab>** character.

-d *char*
Use *char* as the field delimiter instead of the **<tab>** character.

-s Suppress lines with no field-delimiter characters. Unless specified, **cut** passes through unmodified all lines with no delimiters.

cut returns zero on success, one if an error occurred.

Examples

The following example displays all serial port device names found in file **/etc/ttys**.

```
cut -c4- /etc/ttys
```

The following example displays the *login name* and *home directory* fields from the **/etc/passwd** password file. Note that fields in the password file are delimited by the colon character.

```
cut -d: -f1,6 /etc/passwd
```

See Also

awk, commands, paste, sed

Notes

cut is copyright © 1988,1990 by The Regents of the University of California. All rights reserved.

cut is distributed as a service to COHERENT customers, as is. It is not supported by Mark Williams Company. *Caveat utilitor.*

CWD — Environmental Variable

Current working directory

The Korn shell uses the environmental variable **CWD** to hold the current working directory.

See Also

environmental variables, ksh

D

daemon — Definition

A **daemon** is a process that is designed to perform a particular task or control a particular device without requiring the intervention of a human operator.

See Also

definitions

data formats — Technical Information

Mark Williams Company has written C compilers for a number of different computers. Each has a unique architecture and defines data formats in its own way.

The following table gives the sizes, in **chars**, of the data types as they are defined by various microprocessors.

Type	*i8086 SMALL*	*i8086 LARGE*	*Z8001*	*Z8002*	*68000*	*PDP11*	*VAX*
char	1	1	1	1	1	1	1
double	8	8	8	8	8	8	8
float	4	4	4	4	4	4	4
int	2	2	2	2	2	2	4
long	4	4	4	4	4	4	4
pointer	2	4	4	2	4	2	4
short	2	2	2	2	2	2	2

COHERENT places some alignment restrictions on data, which conform to all restrictions set by the microprocessor. Byte ordering is set by the microprocessor; see the Lexicon entry on **byte ordering** for more information.

See Also

byte ordering, C language, data types, double, float, memory allocation, technical information

Notes

The COHERENT system supports Intel SMALL model only.

data types — Technical Information

The following table gives the data types that COHERENT recognizes:

char
double
float
int
long
long float
long int
short
short int
unsigned int
unsigned long int
unsigned long
unsigned char
unsigned short

unsigned short int

The terms **long** and **long int**, as are the terms **short** and **short int**, **double** and **long float**, **unsigned short int** and **unsigned short**, and **unsigned long int** and **unsigned long**. The type **unsigned char** was added to the language by the ANSI Standard; because COHERENT uses signed **char**s by default, you must declare a **char** to be **unsigned** if you want it to be so. If this type is used in arithmetic expressions, it is automatically cast to **unsigned int**.

See Also

C language, char, data formats, double, float, int, long, pointer, short, technical information, unsigned

date — Command

Print/set the date and time
date [-s] [-u] [[*yymmdd*]*hhmm*[.*ss*]]

date prints the time of day and the current date, including the time zone. If an argument is given, the system's current time and date is changed, as follows:

yy	Year (00-99)
mm	Month (01-12)
dd	Day (01-31)
hh	Hour (00-23)
mm	Minute (00-59)
ss	Seconds (00-59)

The seconds fields are optional. For example, typing

```
date 860512141233
```

sets the date to May 12, 1986, and the time to 2:12:33 P.M. At least *hh* and *mm* must be specified — the rest are optional.

The date may be changed only by the superuser.

If option **-s** is specified, **date** suppresses daylight savings time conversion when setting the time.

If option **-u** is specified, dates are set and printed in Greenwich Mean Time (GMT) rather than in local time.

The library time conversion routines used by **date** look for the environmental variable **TIMEZONE**, which specifies local time zone and daylight saving time information in the format described in **ctime**.

See Also

ATclock, commands, ctime(), time, TIMEZONE

Notes

Note that the COHERENT version of the **date** command differs from the UNIX version in that the last two fields of its output are reversed. For example, the UNIX output of **date** reads

```
Sun Jan 13 12:02:09 CST 1991
```

where the COHERENT output reads:

```
Sun Jan 13 12:02:09 1991 CST
```

This may be important when importing UNIX shell commands into COHERENT.

db — Command

Assembler-level symbolic debugger
db [-cdefort] [*mapfile*] [*datafile*]

db is an assembly language-level debugger. It allows you to run object files and executable programs under trace control (see **ptrace**), run programs with embedded breakpoints, and dump and patch files in a variety of forms. You can use it to debug assembly-language programs that have been assembled by **as**, the Mark Williams assembler, as well as those that have been compiled with the Mark Williams C compiler.

What is db?

db is a symbolic debugger, which means that it works with the symbol tables that the compiler builds into the object files it generates. Because **db** is designed to work on the level of assembly language, the user needs a working knowledge of the appropriate assembly language and microprocessor architecture.

Invoking db

To invoke **db**, type its name, plus the options you want (if any) and the name of the files with which you will be working. *mapfile* is an object file that supplies a symbol table. *datafile* is the executable program to be debugged. If both names are given, the options default to **-c**. If only one name is given, it is the *program*; in this case the options default to **-o**. If both names are omitted, *mapfile* defaults to **l.out** and *program* defaults to **core**. If possible, **db** accesses *datafile* with write permission.

The following options to the **db** command specify the format of *program*:

-c *program* is a core file produced by a user core dump. **db** checks the name of the command that invoked the process that produced the core, against the name of the *mapfile*, if given. Pure segments are read from the *mapfile*.

-d *program* is a system dump. If only one file is mentioned, *mapfile* defaults to **/coherent**.

-e The next argument is an object file; **db** executes it as a child process and passes it the rest of the command line.

-f Map *program* as a straight array of bytes (file).

-o *program* is an object file. If *mapfile* is given, it is another object file that provides the symbol table.

-r Read file only, even though you can write into it. This is used to give a file additional protection.

-t Perform input and output for **db** via **/dev/tty**. This permits the debugging of processes whose standard input or output have been redirected.

Commands and Addresses

db executes commands that you give it from the standard input. A command usually consists of an *address*, which tells **db** where in the program to execute the command; and then the command name and its options, if any.

An address is represented by an *expression*, which can be built out of one or more of the following elements:

- The '.', which represents the current address. When an address is entered, the current address is set to that location. The current address can be advanced by typing **<return>**.
- The name of a register. **db** recognizes the register names **r0** through **r7**, **sp**, and **pc** for the PDP-11; **r0** through **r15** and **pc** for the Z-8001 and Z-8002; and **ax**, **ah**, **al**, **bx**, **bh**, **bl**, **cx**, **ch**, **cl**, **dx**, **dh**, **dl**, **si**, **di**, **bp**, **sp**, **sp**, **pc**, **cs**, **ds**, **es**, and **ss** for the i8086. Typing the name of a register displays its contents. The usual numeric base (octal on the PDP-11, hexadecimal on all other machines) is always used for register display and stack tracebacks, regardless of the current default radix.
- The symbols **d**, **i**, and **u**, which represent location 0 in, respectively, the data space, the instruction space, and the u-area.
- The names of global symbols and symbolic addresses can be used in place of the addresses where they occur. This is useful when setting a breakpoint at the beginning of a subroutine.
- An integer constant, which can be used in the same manner as a global symbol. The default is decimal; a leading **0** indicates octal and **0x** indicates hexadecimal.
- The following binary operators can be used:

+	Addition
-	Subtraction
*	Multiplication
/	Integer division

  All arithmetic is done in **longs**.
- The following unary operators can be used:

~	Complementation
-	Negation
*	Indirection

  All operators are supported with their normal level of precedence. Parentheses '()' can be used for binding.

Every symbol refers to a segment: the data segment, the instruction segment, or the u-area. This segment, in turn, dictates the format in which **db** displays by default what it finds at that address. The format used by an expression is that of its leftmost operand. The symbols **d**, **i**, and **u** can name specific segments in the absence of other symbols.

Display Commands

The following commands merely display information about *program*. The symbol '.' represents the *address*, which defaults to the current display address if omitted. *count* defaults to one.

address[,count]?[format]

Display the *format count* times, starting at *address*. The *format* string consists of one or more of the following characters:

^	Reset display address to '.'
+	Increment display address
-	Decrement display address
b	Byte
c	**char**; control and non-**char**s escaped
C	Like 'c' except '\0' not displayed

d	Decimal
f	**float**
F	**double**
i	Machine instruction, disassembled
l	**long**
n	Output '\n'
O	octal
p	Symbolic address
s	String terminated by '\0', with escapes
S	String terminated by '\0', no escapes
u	**unsigned**
W	word
x	Hexadecimal
Y	time (as in i-node etc.)

The format characters **d**, **o**, **u**, and **x**, which specify a numeric base, can be followed by **b**, **l**, or **w**, which specify a datum size, to describe a single datum for display. A format item may also be preceded by a count that specifies how many times the item is to be applied. Note that *format* defaults to the previously set format for the segment (initially **o** for data and u-area, and **i** for instructions). Except where otherwise noted, **db** increments the display address by the size of the datum displayed after each format item.

Execution Commands

In the following commands, *address* defaults to the address where execution stopped, unless otherwise specified; *count* and *expr* default to 1. *commands* is an arbitrary string of **db** commands, terminated by a newline. A newline may be included by preceding it with a backslash '\'.

[address]=
: Print *address* in octal. *address* defaults to '.'. The command = assigns values to locations in the traced process. The size of the assigned value is determined from the last display format used. You can set and display the registers of the traced process, just like any other address in the traced process.

[address[,count]]=value[,value[,value]...]
: Patch the contents starting at *address* to the given *value*. *address* defaults to '.'. Up to ten *values* can be listed.

?
: Print verbose version of last error message.

[address] **:a**
: Print *address* symbolically. *address* defaults to '.'.

*[address]***:b***[commands]*
: Set breakpoint at *address*; save *commands* to be executed when breakpoint is encountered. *commands* defaults to **.:a\ni+.?i\n:x**.

:br *[commands]*
: Set breakpoint at return from current routine. The defaults are the same as for **:b**, above.

[address] **:c**
: Continue execution from *address*.

[address] **:d[r][s]**
: Delete breakpoint at *address*. If optional **r** or **s** is specified, delete return or single-step breakpoint. *address* defaults to '.'.

[address]:**e***[commandline]*
Begin traced execution of the object file at *address* (default, entry point). The *commandline* is parsed and passed to the traced process. **argv[0]** must be typed directly after **:e** if supplied. For example, **:e3 foo bar baz** sets **argv[0]** to **3**, **argv[1]** to **foo**, **argv[2]** to **bar**, and **argv[3]** to **baz**. Quotation marks, apostrophes, and redirection are parsed as by **sh**, but special characters '?*[]' and shell punctuation '()&|;' are not. For complete shell command line parsing use the **-e** option.

Note that the **:e** command must be used to start the program in order to single step, trace back, or display registers. For example, the following command sequence sets a breakpoint at **main**, begins execution, and single-steps through the program:

```
main_:b
:e
:s
```

:f Print type of fault which caused core dump or stopped the traced process.

:m Display segmentation map.

[expr] **:n**
Set default numeric display base to *expr*: **8**, **10**, and **16** indicate, respectively, octal, decimal, and hexadecimal.

:p Display breakpoints.

[expr] **:q**
If *expr* is nonzero, quit the current level of command input (see **:x**). *expr* defaults to 1. End of file is equivalent to **:q**.

:r Display registers.

[address],[count]:**s[c]***[commands]*
Single-step execution starting at *address*, for *count* steps, executing *commands* at each step. *commands* defaults to **i+.?i**.

After a single-step command, **<return>** is equivalent to **.,1:s[c]**. If the optional **c** is present, **db** turns off single-stepping at a subroutine call and turns it back on upon return.

[depth] **:t**
Print a call traceback to *depth* levels. If *depth* is 0 (default), unwind the whole stack.

[expr] **:x**
If *expr* is nonzero, read and execute commands from the standard input up to end of file or **:q**. *expr* defaults to 1.

Please note that the **:c**, **:s**, **:t**, and **:r** commands cannot be executed before a program is started. If you are debugging the program **hello**, do the following first:

```
db hello
main:b
:e
```

This invokes the debugger for **hello** and advances it to **main**. Now you can use the full set of commands.

See Also

commands, core, l.out.h, od, ptrace()

dc — Command

Desk calculator
dc [*file*]

dc is an arbitrary precision desk calculator. It simulates a stacking calculator with ancillary registers. Input must be entered in reverse Polish notation. **dc** maintains the expected number of decimal places during addition, subtraction, and multiplication, but the user must make an explicit request to maintain any places at all during division.

dc reads input from *file* if specified, and then from the standard input. **dc** accepts an arbitrary number of commands per line; moreover, spaces need not be left between them.

The *scale factor* of a number is the number of places to the right of its decimal point. The *scale factor register* controls decimal places in calculations. The scale factor does not affect addition or subtraction. It affects multiplication only if the sum of the scale factors of the two operands is greater than it. The result of every division command has as many decimal places as it specifies. It affects exponentiation in that multiplication is performed as many times as the integer part of the exponent indicates; any fractional part of the exponent is ignored.

dc recognizes the following commands and constructions:

number
: Stack the value of *number*. A number is a string of symbols taken from the digits '0' through '9', and the capital letters 'A' through 'F' (usual hexadecimal notation), with an optional decimal point. An underscore '_' as a prefix indicates a negative number. The letters retain values ten through 15, respectively, regardless of the base chosen by the user.

+ - / * % ^
: The arithmetic operations: addition(+), subtraction(-), division(/), multiplication(*), remainder(%), and exponentiation(^). **dc** pops the two top stack elements, performs the desired operation by calling the multiprecision routine desired (see **multiprecision arithmetic**), and stacks the result.

c Clear the stack.

d Duplicate the top of the stack (so that it occupies the top two positions of the stack).

f Print the contents of the stack and the values of all registers.

i Remove the top of the stack and use its integer part as the assumed input base (default, ten). The new input base must be greater than one and less than 17.

I Stack the current assumed input base.

k Remove the top of the stack and put it in the internal scale factor register.

K Put the value of the internal scale register (which the **k** command sets) on the top of the stack.

l *x* Load the value of register *x* to the top of the stack. The value of register *x* is unaltered. *x* may be any character.

o Remove the top of the stack and use its integer part as the assumed output base (default, ten). The specified base may be any positive integer.

O Stack the current assumed output base.

p Print the top of the stack. The value remains on the stack.

q Quit the program; control returns to the shell **sh**.

s *x* Remove the top of the stack and store it in register *x*. The previous contents of *x* are overwritten. *x* may be any character.

v Replace the top of the stack by its square root.

x Remove the top of the stack, interpret it as a string containing a sequence of **dc** commands, and execute it.

X Replace the top of the stack by its scale factor (i.e., the number of decimal places it has).

z Place the number of occupied levels of the stack on top of the stack.

[...] Place the bracketed character string on top of the stack. The string may be executed subsequently with the **x** command.

<x >x =x !<x !>x !=x
Remove the top two elements of the stack and compare them. If there is no '!' sign before the relation, execute register *x* if the two elements obey the relation. If a '!' sign is present, execute register *x* if the elements do not obey the relation.

! Interpret the rest of the line as a command to the shell **sh**. Control returns to **dc** after command execution terminates.

Example

The following example program prints the first 20 Fibbonacci numbers. The characters **1** and **l** are printed in boldface to help you tell them apart.

```
1sa1sb1sc
[lalbdsa+psblc1+dsc21<y]sy
lyx
```

See Also

bc, commands

Notes

For most purposes the infix notation of **bc** is more convenient than the Polish notation of **dc**.

dcheck — Command

Check directory consistency
dcheck [-s] [-i *inumber...***]** *filesystem ...*

dcheck checks the consistency of each *filesystem*. It scans all the directories in each *filesystem* and counts all i-nodes referenced. It then compares its counts against the link counts maintained in the i-nodes. **dcheck** notes any discrepancies, and notes allocated i-nodes with a link count of zero.

The **-i** argument tells **dcheck** to compare each *inumber* in the list against those in each directory. It reports matches by printing the i-number, the i-number of the parent directory, and the name of the entry. The **-s** argument tells **dcheck** to correct the link count

of errant i-nodes to the entry count.

Because **dcheck -s** uses two passes to check a *filesystem*, the file system should be unmounted. If **-s** is used on the root file system, the system should be rebooted immediately (without performing a **sync**). The raw device should be used.

See Also

check, commands, dir.h, icheck, ncheck, sync, umount

Diagnostics

If the link count is zero and there are entries, the file system must be mounted and all entries removed immediately. If the link count is nonzero and the entry count is *larger*, the **-s** option must be used to make the counts agree. In all other cases there may be wasted disk space but there is no danger of losing file data.

Notes

In earlier releases of COHERENT, **dcheck** acted upon a default file system if none was specified.

This command has largely been replaced by **fsck**.

dd — Command

File conversion
dd [*option=value*] **...**

dd copies an input file to an output file, while performing requested conversions. Options include case and character set conversions, byte swapping conversion for other machines, and different input and output buffer sizes. **dd** can be used with raw disk files or raw tape files to do efficient copies with large block (record) sizes. Read and write requests can be changed with the **bs** option described below.

The following list gives each available *option*. Any numbers which specify block sizes or seek positions may be written in several ways. A number followed by **w**, **b**, or **k** is multiplied by two (for words), 512 (for blocks), or 1,024 (for kilobytes), respectively, to obtain the size in bytes. A pair of such numbers separated by **x** is multiplied together to produce the size. All buffer sizes default to 512 bytes if not specified.

bs=*n* Set the size of the buffer for both input and output to *n* bytes. blocks??

cbs=*n* Set the conversion buffer size to *n* bytes (used only with character set conversions between ASCII and EBCDIC).

conv=*list* Perform conversions specified by the comma-separated *list*, which may include the following:

ascii	Convert EBCDIC to ASCII
ebcdic	Convert ASCII to EBCDIC
ibm	Convert ASCII to EBCDIC, IBM flavor
lcase	Convert upper case to lower
noerror	Continue processing on I/O errors
swab	Swap every pair of bytes before output
sync	Pad input buffers with 0 bytes to size of **ibs**
ucase	Convert lower case to upper

count=*n* Copy a maximum of *n* input records.

files=*n* Copy a maximum of *n* input files (useful for multifile tapes).

ibs=*n* Set the input buffer size to *n* (normally used if input and output blocking sizes are to be different).

if=*file* Open *file* for input; the standard input is used when no **if=** option is given.

obs=*n* Set the output buffer size to *n*.

of=*file* Open *file* for output; the standard output is used when no **of=** option is given.

seek=*n* Seek to position *n* bytes into the output before copying (does not work on stream data such as tapes, communications devices, and pipes).

skip=*n* Read and discard the first *n* input records.

Examples

The first example copies the entire contents of a 1.44-megabyte, 3.5-inch diskette from drive 0 to file **disk.dd**:

```
dd if=/dev/fva0 of=disk.dd bs=36b count=80
```

The second example writes the contents of the previously stored 5.25-inch file **backup.dd** to a 1.2-megabyte, 5.25-inch floppy disk in drive 1:

```
dd if=backup.dd of=/dev/fha1 bs=30b count=80
```

See Also

ASCII, commands, conv, cp, tape, tr

Diagnostics

The command reports the number of full and partial buffers read and written upon completion.

Notes

Because of differing interpretations of EBCDIC, especially for certain more exotic graphic characters such as braces and backslash, no one conversion table will be adequate for all applications. The **ebcdic** table is the American Standard of the Business Equipment Manufacturers Association. The **ibm** table seems to be more practical for line printer codes at many IBM installations.

default — C Keyword

Default label in switch statement

default is a prefix used in **switch** statement. If none of the **case** labels match the parameter in the **switch** statement, then the **default** label is used. A **switch** is not required to have a **default** case, but it is good programming practice to use one.

See Also

C keywords, case, switch

definitions — Overview

The Lexicon contains the following articles that define aspects of COHERENT:

address	**alignment**	**arena**
array	**bit**	**bit map**
buffer	**byte**	**cast**
caveat utilitor	**cc0**	**cc1**
cc2	**cc3**	**daemon**

directory
file
filter
i-node
macro
named pipe
nybble
pattern
process
ranlib
root
standard error
stderr
sticky bit
superuser
executable file
FILE
function
interrupt
manifest constant
NUL
object format
pipe
pun
read-only memory
rvalue
standard input
stdin
stream
wildcards
field
file descriptor
GMT
lvalue
modulus
NULL
operator
port
random access
register variable
stack
standard output
stdout
structure

See Also

Lexicon

deftty.h — Header File

Define default tty settings
#include <sys/deftty.h>

deftty.h defines the default tty settings.

See Also

header files

deroff — Command

Remove text formatting control information
deroff [-w] [-x] [*file* ...]

deroff removes text formatting control information from each input text *file*, or from the standard input if no *file* is specified. It regards all lines that begin with '.' or '" as being **nroff** or **troff** commands and deletes them. **deroff** also recognizes some additional control lines. It deletes **eqn** information (between **.EQ** and **.EN** lines), **tbl** information (between **.TS** and **.TE** lines), and macro definitions. It also deletes embedded **.eqn** requests. It expands source file inclusion with **.so** and **.nx** requests, with the proviso that no input file is read twice. It also deletes some **troff** escape sequences, such as those for font and size change.

When the **-x** flag is present, **deroff** uses some additional knowledge about the **nroff -ms** macro package.

When the **-w** flag is present, **deroff** divides the remaining text into words and prints them to the standard output, one per line. A **word** comprises a sequence of letters, digits, and apostrophes that commences with a letter. **deroff** strips apostrophes from the output. All other characters between words are not printed. The spelling checking programs **spell** and **typo** use this option.

See Also

commands, nroff, spell, troff, typo

detab — Command

Replace tab characters with spaces
detab [-t*tabsize*]

detab reads the standard input, replaces every tab character with spaces, and writes the result to the standard output. If you do not specify the **-t** option, **detab** uses the standard value of eight. *tabsize* can range in value from two to 256, inclusive.

See Also

commands

device drivers — Overview

A *device driver* is a program that controls the action of one of the physical devices attached to your computer system.

The following table lists the device drivers included with this edition of the COHERENT system. The first field gives the device's major device number; the second gives its name; and the third describes it. When a major device number has no driver associated with it, that device is available for a driver yet to be written.

0:	***mem**	Interface to memory
1:	**tty**	Primitive tty driver
2:	**nkb/kb/mm**	Keyboard and video
3:	**lp**	Parallel line printer
4:	**fl**	Floppy drive
5:	**al0**	Serial line 0 (COM1 and COM3)
6:	**al1**	Serial line 1 (COM2 and COM4)
7:	**hs**	Generic polled multi-port serial card
8:	**rm**	Dual RAM disk
9:		
10:		
11:	**at**	AT hard disk
12:		
13:	**scsi**	SCSI device drivers: **aha154x**, **ss**
14:		
15:		
16:		
17:		
18:		
19:		
20:		
21:		
22:		
23:	**sem**	System V compatible semaphores
24:	**shm**	System V subset shared memory
25:	**msg**	System V compatible messaging
26:		
27:		
28:		
29:		
30:		
31:		

Also included are drivers for the following devices:

console	Console driver
ct	Controlling terminal driver
null	The "bit bucket"
sgtty	Terminal driver

Major and Minor Numbers

COHERENT uses a system of *major* and *minor* device numbers to manage devices and drivers. In theory, COHERENT assigns a unique major number to each type of device, and a unique minor number to each instance of that type. In practice, however, a major number describes a device driver (rather than a device *per se*). Each device driver uses one or more unique major numbers, and the individual devices serviced by that driver are identified by a minor number. There are, however, a number of exceptions to this scheme:

1. Sometimes, certain parts of the minor number specify configuration. For example, bits 0 through 6 of the minor number for COHERENT RAM disks indicate the size of the allocated device.

2. In COHERENT, devices using different IRQ's may have different major numbers, even if the devices are of the same general type. For example, devices **com1*** and **com3*** have major number 5, while **com2*** and **com4*** have major number 6.

See Also

at, boot, com, console, ct, fl, Lexicon, lp, mboot, mem, msg, null, sem, sgtty, shm, tape

Notes

See the Release Notes for your release of COHERENT for a list of supported devices and device drivers.

The devices **msg**, **sem**, and **shm** are loadable drivers that can be loaded into memory using the command **drvld**. See their respective entries in the COHERENT Lexicon for more information.

df — Command

Measure free space on disk
df [-ait] *device*

df measures the amount of free space left on a floppy disk, on a logical device on a hard disk, or on a RAM disk. *device* is the name of the device you wish to check. For example, to check the amount of space left on filesystem **x**, type:

```
df /x
```

The default device is the one you are currently using. **df** displays three numbers: the total number of disk blocks in the device, the number of disk blocks being used, and the percent of total disk blocks that are free. Note that a disk block is 512 bytes (1/2 kilobyte).

df recognizes the following options:

-a Prints the amount of space left on all devices.

-i Show the number and percentage of i-nodes available.

-t Show the total disk space available.

See Also

commands

diff — Command

Summarize differences between two files
diff [-bdefh] [-c *symbol*] *file1 file2*

diff compares *file1* with *file2*, and prints a summary of the changes needed to turn *file1* into *file2*.

Two options involve input file specification. First, the standard input may be specified in place of a file by entering a hyphen '-' in place of *file1* or *file2*. Second, if *file1* is a directory, **diff** looks within that directory for a file that has the same name as *file2*, then compares *file2* with the file of the same name in directory *file1*.

The default output script has lines in the following format:

```
1,2 c 3,4
```

The numbers *1,2* refer to line ranges in *file1*, and *3,4* to ranges in *file2*. The range is abbreviated to a single number if the first number is the same as the second. The command *c* was chosen from among the **ed** commands '*a*', '*c*', and '*d*'. **diff** then prints the text from each of the two files. Text associated with *file1* is preceded by '< ', whereas text associated with *file2* is preceded by '>'.

The following summarizes **diff**'s options.

-b Ignore trailing blanks and treat more than one blank in an input line as a single blank. Spaces and tabs are considered to be blanks for this comparison.

-c *symbol*
Produce output suitable for the C preprocessor **cpp**; the output contains **#ifdef**, **#ifndef**, **#else**, and **#endif** lines. *symbol* is the string used to build the **#ifdef** statements. If you define *symbol* to the C preprocessor **cpp**, it will produce *file2* as its output; otherwise, it will produce *file1*. This option does *not* work for files that already contain **#ifdef**, **#ifndef**, **#else**, and **#endif** statements.

-e Create an **ed** script that will convert *file1* into *file2*.

-f Produce a script in the same manner as the **-e** option, but with line numbers taken directly from the two input files. This will work properly only if applied from end to beginning; it cannot be used directly by **ed**.

-h Compare large files that have a minimal number of differences. This option uses an algorithm that is not limited by file length, but may not discover all differences.

-d Select the **-h** algorithm only for files larger than 25,000 bytes; otherwise, use the normal algorithm.

See Also

ed, egrep, commands

Diagnostics

diff's exit status is zero when the files are identical, one when they are different, and two if a problem was encountered (e.g., could not open a file).

Notes

diff cannot handle files with more than 32,000 lines. Handing **diff** a file that exceeds that limit will cause it to fail, with unpredictable side effects.

diff3 — Command

Summarize differences among three files
diff3 [-ex3] *file1 file2 file3*

diff3 summarizes the differences among three text files. Each difference encountered is headed by one of the following separators, which categorizes how many of the three input files differ in a given range. The headers are as follows

====
All of the files are different.

====*n*
Only the *n*th file differs, where *n* may be 1, 2, or 3.

For each set of changes marked as above, the actual change is indicated for each file using a notation similar to commands to **ed**. For each *filen* the following is printed:

*n: l***a** Text is to be appended after line *l* in *filen*.

*n: l,m***c** The text from line *l* to line *m* is to be changed for *filen*. The original text from *filen* follows this line. If this text is identical for two of the files, only the latter (higher numbered) of the two is printed.

Options are available to print a script of commands to **ed**. With the **-e** option, a script that will make all changes between *file2* and *file3* to *file1* is produced. This script is based upon all changes flagged with **====** or **====3** separators, as described above.

The **-x** option prints only those changes where all three files differ, i.e., those flagged with **====**.

The **-3** option requests only those changes where *file3* differs.

Example

The following command sequence produces a script, applies it to *file1*, and sends the result to the standard output.

```
(diff3 -e file1 file2 file3; echo '1,$p') | ed - file1
```

Files

/tmp/d3*
/usr/lib/diff3

See Also

commands, diff, ed

Diagnostics

An exit status of zero indicates all three files were identical, one indicates differences, and two indicates some other failure.

dir.h — Header File

Directory format
#include <dir.h>

A COHERENT directory is exactly like an ordinary file, except that a user's process may write on it only through system calls such as **creat**, **link**, **mknod**, or **unlink**. The system

distinguishes directories from other types of files by the mode word **S_IFDIR** in the i-node. (For more information on i-nodes, see **stat**).

Every directory is an array of entries of the following structure, as defined in the header file **dir.h**:

```
#define                         DIRSIZ 14

struct direct {
        ino_t d_ino;            /* i-number */
        char d_name[DIRSIZ];    /* name */
};
```

Any entry in which **d_ino** has a value of zero is unused.

The command **mkdir** creates a directory, with the convention that its first two entries are '.' and '..'. The name '.' is self-referential — a link to the directory itself. The name '..' is a link to the parent directory. Because the root directory has no parent, its '..' is a link to itself.

The **d_ino** entry of the directory structure is stored in the file system in canonical form, as described in **canon.h**.

See Also

canon.h, header files, mkdir, stat()

directory — Definition

A **directory** is a table that maps names to files; in other words, it associates the names of a file with their locations on the mass storage device. Under some operating systems, directories are also files, and can be handled like a file.

Directories allow files to be organized on a mass storage device in a rational manner, by function or owner.

See Also

definitions, file

dirent.h — Header File

Define dirent
#include <dirent.h>

dirent.h defines the manifest constant **dirent**.

See Also

header files

dirs — Command

Print the contents of the directory stack
dirs

The COHERENT shell **sh** maintains an internal "directory stack", which is a stack of names of directories. You can manipulate this stack should you, for any reason, wish to traverse a number of directories quickly and efficiently.

The command **dirs** prints the current contents of the directory stack.

See Also

commands, popd, pushd, sh

disable — Command

Disable a port
/etc/disable *port...*

disable tells the COHERENT system not to create a login process for each given asynchronous *port*. For example, the command

```
/etc/disable com1r
```

disables port **/dev/com1r**. **disable** changes the entry for each given *port* in the terminal characteristics file **/etc/ttys**, and signals **init** to rescan the **ttys** file.

The command **enable** enables a port. The command **ttystat** checks whether a port is enabled or disabled.

Files

/etc/ttys — Terminal characteristics file

See Also

com, commands, enable, login, ttys, ttystat

Diagnostics

disable normally returns one if it disables the *port* successfully and zero if not. If more than one *port* is specified, **disable** returns the success or failure status of the last port it finds. It returns -1 if it cannot find any given *port*. An exit status of -2 indicates an error.

Notes

Only the superuser **root** can execute **disable**.

div() — General Function (libc)

Perform integer division
#include <stdlib.h>
div_t div(*numerator, denominator***)**
int *numerator, denominator***;**

div divides *numerator* by *denominator*. It returns a structure of the type **div_t**, which is structured as follows:

```
typedef struct {
        int quot;
        int rem;
} div_t;
```

div writes the quotient into **quot** and the remainder into **rem**.

The sign of the quotient is positive if the signs of the arguments are the same; it is negative if the signs of the arguments differ. The sign of the remainder is the same as the sign of the numerator.

If the remainder is non-zero, the magnitude of the quotient is the largest integer less than the magnitude of the algebraic quotient. This is not guaranteed by the operators / and %, which merely do what the machine implements for divide.

See Also

general functions, ldiv

Notes

The ANSI Standard includes this function to permit a useful feature found in most versions of FORTRAN, where the sign of the remainder will be the same as the sign of the numerator. Also, on most machines, division produces a remainder. This allows a quotient and remainder to be returned from one machine-divide operation.

If the result of division cannot be represented (e.g., because *denominator* is set to zero), the behavior of **div** is undefined. *Caveat utilitor.*

do — C Keyword

Introduce a loop

do is a C control statement that introduces a loop. Unlike **for** and **while** loops, the condition in a **do** loop is evaluated *after* the operation is performed. **do** always works in tandem with **while**; for example

```
do {
     puts("Next entry? ");
     fflush(stdout);
} while(getchar() != EOF);
```

prints a prompt on the screen and waits for the user to reply. The **do** loop is convenient in this instance because the prompt must appear at least once on the screen before the user replies.

See Also

break, C keywords, continue, while

dos — Command

Transfer files to/from an MS-DOS file system
dos [-]dFlrtx[*flags*] [*device*] [*file* ...]

The command **dos** allows the COHERENT user to manipulate an MS-DOS file system, which may be either a hard-disk partition or a floppy disk. It can build an empty MS-DOS file system, label it, list the files in it, transfer files between it and COHERENT, or delete files from it.

The given *device* must be a special file that specifies an MS-DOS file system, such as floppy-disk drive **/dev/fha0** or hard-disk partition **/dev/at0a**. The default *device* is **/dev/dos**, which the system administrator should link to the most commonly used device name.

dos converts between the differing file-name conventions of COHERENT and MS-DOS. An MS-DOS *file* argument may be specified in lower or upper case, using '/' as the path-name separator. When transferring files from MS-DOS to COHERENT, **dos** converts an MS-DOS file name to a COHERENT file name in lower case only. If the MS-DOS file name contains no extension, the COHERENT file name contains no '.'. When transferring files from COHERENT to MS-DOS, **dos** converts all alphabetic characters in a COHERENT file name to upper case; if a period '.' appears at the beginning or end of a file name, **dos** converts it to '_'. **dos** truncates the part of the file name before the last '.' to a maximum of eight characters and truncates the extension to a maximum of three characters.

The command line must specify exactly one of the following *functions*.

d Delete each *file* from the MS-DOS file system. This option also allows the user to delete empty directories.

F Create an empty MS-DOS file system on a formatted diskette. This option is analogous to the COHERENT command **/etc/mkfs**. The COHERENT commands **/etc/fdformat** and **/etc/mkfs** initialize a COHERENT diskette in two steps. The MS-DOS command **format** initializes an MS-DOS diskette by performing both the physical and logical formatting operations with one command. To initialize an MS-DOS diskette under COHERENT, use the command **/etc/fdformat -v** *devicename*, followed by the command **dos F** *devicename*. If *file* is named, **dos** copies it to the boot block of the file system. The **dos** command cannot build a file system on a hard-disk partition.

l Label the MS-DOS file system. The command line must specify exactly one *file* argument, which gives the label.

r Replace each *file* on the MS-DOS file system with the COHERENT file of the same name. If a given *file* argument specifies a COHERENT directory, **dos** replaces its subdirectories recursively to the MS-DOS file system unless the **s** flag is used. If no *file* is specified, **dos** copies all files in the current directory to the MS-DOS file system.

t List the files on the MS-DOS file system. If no *file* argument is given, **dos** lists the entire MS-DOS file system; otherwise, it lists each *file*. If a *file* argument specifies an MS-DOS subdirectory, **dos** lists its contents. **dos** lists directories first in alphabetical order, then ordinary files in alphabetical order.

x Extract each *file* from the MS-DOS file system to a COHERENT file of the same name. If a given *file* argument specifies an MS-DOS subdirectory, **dos** extracts its contents recursively unless the **s** flag is used. If no *file* is given, **dos** extracts all files from the MS-DOS file system to the current COHERENT directory.

The following *flags* are available.

a Perform ASCII newline conversion on file transfer. When moving files from COHERENT to MS-DOS, this option converts each COHERENT newline character '\n' (ASCII **LF**) to an MS-DOS end-of-line (ASCII **CR** and **LF**); when moving files from MS-DOS to COHERENT, it does the opposite. By default, **dos** performs binary file transfer, without newline conversion.

k Keep the file modification time (mtime) on extract and replace operations. By default, **dos** gives extracted or replaced files the current time. With this option, **dos** gives the extracted or replaced file the same time as the original file.

n List files in order of creation (newest file last) rather than in alphabetical order. This option applies only to the table-of-contents function. **dos** always lists directories before files, with or without the **n** option.

p Perform a piped extract or replace (for use in pipelines). The command line must specify exactly one *file* argument. For extract, **dos** reads the given *file* and writes it to the standard output. For replace, **dos** reads the standard input and writes it to the given *file*.

s Suppress extraction or replacement of subdirectories. By default, **dos** extracts or replaces subdirectories recursively.

v Verbose option. Provide additional information about each function performed.

[1-9] A digit specifies a logical drive number on an extended MS-DOS partition. For example, **dos tv2 /dev/at0c** lists the directory of the second logical drive on extended MS-DOS partition **/dev/at0c**.

Examples

The first example copies all files located in directories **sources** and **include**, as well as any subdirectories, from floppy drive **/dev/fva1** to correspondingly named subdirectories in the current COHERENT directory:

```
dos xavk /dev/fva1 sources include
```

Note that **fva1** is a high-density, 3.5-inch floppy disk in floppy-disk drive 1 (a.k.a., drive B:). The files will be copied with ASCII newline conversion and will retain the time and date that they had under MS-DOS.

The next example copies a file from an MS-DOS partition on your hard disk. Suppose that **C:** is the primary MS-DOS partition on your first hard drive. The following command copies file **C:\AUTOEXEC.BAT** to **/autoexec.bat** in your COHERENT root partition:

```
dos xa /dev/at0a /autoexec.bat
```

You will want to use the **a** switch any time you are transferring a text file.

Suppose that the second partition on your first hard drive (COHERENT device **/dev/at0b**) is an extended MS-DOS partition with two logical drives, **D:** and **E:**. To copy a COHERENT text file **/tmp/foo** to **D:\TMP\FOO**, use the command

```
dos ra1 /dev/at0b /tmp/foo
```

To copy non-text file **frotz** in the current COHERENT directory to MS-DOS file **E:\DBF\AX\FROTZ**, use the command

```
dos rp2 /dev/at0b dbf/ax/frotz < frotz
```

See Also

commands, fdformat, mkfs, MS-DOS

Notes

dos does not work with MS-DOS hard-disk file systems that hold more than 64,000 clusters (i.e., with four-byte FAT entries rather than 1.5-byte or two-byte FAT entries). It does understand MS-DOS 3.3 extended disk partitions (where a single partition contains more than one MS-DOS file system), but you must know whether a partition contains a normal MS-DOS partition or an extended partition.

dos does not check for unusual characters in a COHERENT file name or for file names that differ from other file names only in case.

double — C Keyword

Data type

A **double** is the data type that encodes a double-precision floating-point number. On most machines, **sizeof(double)** is defined as four machine words, or eight **char**s. If you wish your code to be portable, do *not* use routines that depend on a **double** being 64 bits long.

Different formats are used to encode **double**s on various machines. These formats include IEEE, DECVAX, and BCD (binary coded decimal), as described in the entry for **float**.

/etc/drvld /drv/st
/etc/drvld.all??
mm/dev 666 bin bin
are inodes for char devices there

See Also

C keywords, data formats, float, portability

drvld — Command

Load a loadable driver into memory
/etc/drvld *options driver*

drvld loads a loadable driver into memory. *driver* names a loadable driver. Only the superuser **root** can run **drvld**.

A loadable driver is one that is not linked into the kernel when it was built. The current suite of loadable drivers include multi-port serial cards, various SCSI host adapters, and a variety of add-on cards. The COHERENT drivers for shared memory, semaphores, and message passing are also implemented as loadable drivers, due to the efficient size of the COHERENT kernel.

drvld recognizes the following *options*:

-k *kernel*
: By default, **drvld** assumes that file **/coherent** holds the symbol table for the in-core copy of COHERENT. The **-k** option tells **drvld** to load the driver using a version of COHERENT other than the default. You must use this option if you are running an alternate copy of COHERENT (e.g., a version based on the floppy disk drive).

-r Supress generation of a debugging symbol table.

-o *outfile*
: By default, **drvld** writes the driver's debugging symbol table into a file that has the same name as the driver but is located in directory **/tmp**. The **-o** options tells **drvld** to output the symbol table to *outfile* rather than the default.

Files

/drv — Directory that contains loadable drivers

See Also

commands, device drivers, sload()

Notes

COHERENT supports user-written, loadable device drivers generated with the COHERENT device-driver kit. By convention, loadable drivers that have been tested thoroughly and released for production reside in directory **/drv**, not in **/dev**.

If you see a message of the form "loadable drivers disabled", you have attempted to use **drvld** on a kernel other than **/coherent** without specifying the **-k** option.

du — Command

Summarize disk usage
du [-a] [-s] [*directory* ...]

du prints the total number of disk blocks used by each named *directory*. If no *directory* is specified, **du** prints the disk usage of the current directory.

The **-a** (all) option causes **du** to print a line for every file and directory in the substructure. Normally it prints a line only for each directory.

The **-s** (summary) option prints only the line for the top level directory.

du understands links; it adds a file with more than one link to it into the total only once.

See Also

commands, df, find

Notes

du does not count file-system overhead such as indirect blocks, so occasionally a directory does not fit on a file system which appears to contain enough room for it.

dump — Command

File-system backup utility
dump [*options*] [*argument* ...]

dump dumps either all or a portion of file system *argument* to magnetic tape or floppy disks. File-system dumps are in a format that permits you to restore all or some of the files to the original file system, and to select files either by name or by i-number.

A file-system dump includes all files changed since the *dump since* date, plus each file's full path name (for the benefit of **dumpdir**).

options specifies both the dump-since date and the processing options. It is made up of characters from the set **0123456789bdfsSuv**, which have the following meanings.

0-9 The digit gives the level number of the dump. The dump-since date is the most recent date in the dump-date file **/etc/ddate** that is (1) associated with this file system and (2) has a level number less than the current dump level. For example, if you request a level-3 dump, **dump** will back up all files not backed up since the last level-2 dump. A level-0 dump by definition backs up all files in the file system.

b The next *argument* gives the output tape's *blocking factor*. The blocking factor is the number of **dumpdata** structures in each tape block. The default blocking factor is 20.

d The next *argument* gives the density of the output tape in bytes per inch. The default density is 1600 bytes per inch (bpi). **dump** uses the density to compute the quantity of tape needed.

f The next *argument* gives the path name of the output file. If no **f** option is given, **/dev/dump** is assumed.

s The next *argument* gives the length of the dump tape in feet. **dump** keeps a running total of the quantity of tape it has written, and it asks for a new reel if it appears that the end of the reel is near. The default length is 2,300 feet.

S The next *argument* gives the size of the dump output device, in blocks. This is used only if you are backing up the file system to floppy disks or streaming cartridge tape rather than to nine-track magnetic tape.

u If the dump completes without error, update the record of successful dumps kept in file **/etc/ddate**. There is an entry in this file for every file system and every dump level.

v Inform the user of the 'dump since' date and the length of tape used in feet. The length is useful for computing the quantity of tape remaining if multiple dumps are written onto a single reel of tape.

If no level number is given, **dump** assumes the *options* **9u**.

Files

/dev/dump — Default dump device
/etc/ddate — Dump date file

See Also

badscan, commands, dumpdate, dumpdir, restor

Diagnostics

Most errors are fatal, caused by a table overflowing or a read or write error on the input or output device.

dump requires that its output be written to disks that are free of bad sectors. If you write a dump to a disk with bad sectors, you will not be able to restore files from that disk.

When formatting disks to be used with **dump**, use the command

```
/etc/fdformat -v device
```

This forces **fdformat** to verify the format. It takes twice as long, but it ensures that the disk is good at least at a first level of testing. Reject any disks that have any defects — or save them for use with COHERENT file systems, which can map out bad sectors.

dumpdate — Command

Print dump dates
dumpdate [*filesystem* ...]

dumpdate reads through the dump date file **/etc/ddate** and displays the dump date records associated with each specified *filesystem*.

If no *filesystem* is specified, the records for all file systems are displayed.

Files

/etc/ddate — Dump date file

See Also

commands, dump, dumpdir, restor

dumpdir — Command

Print the directory of a dump
dumpdir [**af** [*argument* ...]]

dumpdir reads through a file-system dump created by the **dump** command, gathers up its directory blocks, and displays the names and i-numbers of all files on the dump.

The **a** option causes **dumpdir** to display the directory entries for '.' and '..', which are normally suppressed.

The **f** option causes the next *argument* to be taken as the pathname of the dump device, which is otherwise assumed to be **/dev/dump**.

If no options are specified, **dumpdir** reads from the default dump device **/dev/dump** and suppresses the printing of '.' and '..' entries.

Files

/dev/dump — Default dump device
/tmp/dd*XXXXXX* — To hold directory blocks

See Also

commands, dump

Diagnostics

The dump/restore format puts a header at the beginning of the dump that includes all the information about what lives where in the dump. **dumpdir** reads this header to discover what files are in the dump. If the header is too large to fit onto one disk, **dumpdir** will then prompt you to insert the additional disk or disks; if this happens, insert the requested disk and then type **<return>**.

Notes

dump requires that its output be written to disks that are free of bad sectors. If you write a dump to a disk with bad sectors, you will not be able to restore files from that disk. For details on using disks with **dump**, see its Lexicon entry.

dumptape.h — Header File

Define data structures used on dump tapes
#include <dumptape.h>

dumptape.h defines the data structures used on dump tapes. A dump tape begins with a header record. This contains the attributes of the tape. The remainder of the tape is filled with arrays of dumpdata records. The map comes first, then all the directories, then all the files.

See Also

dump, header files

dup() — COHERENT System Call (libc)

Duplicate a file descriptor
int dup(*fd***) int** *fd***;**

dup duplicates the existing file descriptor *fd*, and returns the new descriptor. The returned value is the smallest file descriptor that is not already in use by the calling process.

See Also

COHERENT system calls, dup2(), fopen(), fdopen(), STDIO

Diagnostics

dup returns a number less than zero when an error occurs, such as a bad file descriptor or no file descriptor available.

dup2() — COHERENT System Call (libc)

Duplicate a file descriptor
int dup2(*fd***,** *newfd***) int** *fd***,** *newfd***;**

dup2 duplicates the file descriptor *fd*. Unlike its cousin **dup**, **dup2** allows you to specify a new file descriptor *newfd*, rather than having the system select one. If *newfd* is already open, the system closes it before assigning it to the new file. **dup2** returns the duplicate descriptor.

See Also

COHERENT system calls, dup(), STDIO

Diagnostics

dup2 returns a number less than zero when an error occurs, such as a bad file descriptor or no file descriptor available.

E

ebcdic.h — Header File

Define manifest constants for non-printable EBCDIC characters
#include <ebcdic.h>

ebcdic.h defines manifest constants for non-printable characters used in the EBCDIC character set. The constants correspond to those defined in the header file **ascii.h**.

See Also

ASCII, ascii.h, header files

echo — Command

Repeat/expand an argument
echo [-n] [*argument* ...]

echo prints each *argument* on the standard output, placing a space between each *argument*. It appends a newline to the end of the output unless the **-n** flag is present.

echo recognizes the following special character sequences. For each occurrence of the sequence, it substitutes the correspondingASCII character.

\b	Backspace
\c	Print line without a newline (like **-n** option)
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Tab
\v	Vertical tab
\\	Backslash
nnn	*nnn* is the octal value of the desired character

For example, if you type:

```
echo 'Please enter your name: \007\c'
```

the shell prints

```
Please enter your name:
```

on your screen. Note that the **\007** sequence causes the terminal bell to sound, and that since the **\c** sequence was specified, the cursor will be left positioned after the colon.

See Also

commands, ksh, sh

Notes

Under the Korn shell, **echo** is an alias for its built-in command **print**.

ed — Command

Interactive line editor
ed [-] [+cmopsv] [*file*]

ed is the COHERENT system's interactive line editor.

ed is a line-oriented interactive text editor. With it, you can locate and replace text patterns, move or copy blocks of text, and print parts of the text. **ed** can read text from

input files and can write all or part of the edited text to other files.

ed reads commands from the standard input, usually one command per line. Normally, **ed** does not prompt for commands. If the optional *file* argument is given, **ed** edits the given file, as if the *file* were read with the **e** command described below.

ed manipulates a copy of the text in memory rather than with the file itself. No changes to a file occur until the user writes edited text with the **w** command. Large files can be divided with **split** or edited with the stream editor **sed**.

ed remembers some information to simplify its commands. The *current line* is typically the line most recently edited or printed. When **ed** reads in a file, the last line read becomes the current line. The *current file name* is the last file name specified in an **e** or **f** command. The *current search pattern* is the last pattern specified in a search specification.

ed identifies text lines by integer line numbers, beginning with one for the first line. Several special forms identify a line or a range of lines, as follows:

n A decimal number *n* specifies the *n*th line of the text.

. A period '.' specifies the current line.

**$** A dollar sign '$' specifies the last line of the text.

+,- Simple arithmetic may be performed on line numbers.

/pattern/
Search forward from the current line for the next occurrence of the *pattern*. If **ed** finds no occurrence before the end of the text, the search wraps to the beginning of the text. Patterns, also called *regular expressions*, are described in detail below.

?*pattern***?**
Search backwards from the current line to the previous occurrence of the *pattern*. If **ed** finds no occurrence before the beginning of the text, the search wraps to the end of the text.

'x Lines marked with the **k***x* command described below are identified by *'x*. The *x* may be any lower-case letter.

n,m Line specifiers separated by a comma ',' specify the range of lines between the two given lines, inclusive.

n;m Line specifiers separated by a semicolon ';' specify the range of lines between the two given lines, inclusive. Normally, **ed** updates the current line after it executes each command. If a semicolon ';' rather than a comma separates two line specifiers, **ed** updates the current line before reading the second.

***** An asterisk '*' specifies all lines; it is equivalent to **1,$**.

Commands

ed commands consist of a single letter, which may be preceded by one or two specifiers that give the line or lines to which the command is to be applied. The following command summary uses the notations [**n**] and [**n**[,**m**]] to refer to an optional line specifier and an optional range, respectively. These default to the current line when omitted, except where otherwise noted. A semicolon ';' may be used instead of a comma ',' to separate two line specifiers.

. Print the current line. Also, a line containing only a period '.' marks the end of **a**ppended, **c**hanged, or **i**nserted text.

[*n*] Print given line. If no line number is given (i.e., the command line consists only of a newline character), print the line that follows the current line.

[*n*]= Print the specified line number (default: last line number).

[*n*]& Print a screen of 23 lines; equivalent to **n**,**n**+22*p*.

! *line* Pass the given *line* to the shell **sh** for execution. **ed** prompts with an exclamation point '!' when execution is completed.

? Print a brief description of the most recent error.

[*n*]**a** Append new text after line *n*. Terminate new text with line that contains only a period '.'.

[*n*[,*m*]]**c**
Change specified lines to new text. Terminate new text with a line that contains only a period '.'.

[*n*[,*m*]]**d**[**p**]
Delete specified lines. If **p** follows, print new current line.

e [*file*]
Edit the specified *file* (default: current file name). An error occurs if there are unsaved changes. Reissuing the command after the error message forces **ed** to edit the *file*.

E [*file*]
Edit the specified *file* (default: current file name). No error occurs if there are unsaved changes.

f [*file*]
Change the current file name to *file* and print it. If *file* is omitted, print the current file name.

[**n**[,*m*]]**g**/[*pattern*]/*commands*
Globally execute *commands* for each line in the specified range (default: all lines) that contains the *pattern* (default: current search pattern). The *commands* may extend over several lines, with all but the last terminated by '\'.

[*n*]**i** Insert text before line *n*. Terminate new text with a line that contains only a period '.'.

[*n*[,*m*]]**j**[**p**]
Join specified lines into one line. If *m* is not specified, use range *n*,*n*+1. If no range is specified, join the current line with the next line. With optional **p**, print resulting line.

[*n*]**k***x*
Mark given line with lower-case letter *x*.

[*n*[,*m*]]**l**
List selected lines, interpreting non-graphic characters.

[*n*[,*m*]]**m**[*d*]
Move selected lines to follow line *d* (default: current line).

o *options*
Change the given *options*. The *options* may consist of an optional sign '+' or '-', followed by one or more of the letters '*cmopsv*'. Options are explained below.

[*n*[,*m*]][**p**]
Print selected lines. The **p** is optional.

q Quit editing and exit. An error occurs if there are unsaved changes. Reissuing the command after the error message forces **ed** to exit.

Q Quit editing and exit. No error occurs if there are unsaved changes.

[*n*]**r** [*file*]
Read *file* into current text after given line (default: last line).

[**n**[,*m*]]**s**[*k*]/[*pattern1*]/*pattern2*/[**g**][**p**]
Search for *pattern1* (default, remembered search pattern) and substitute *pattern2* for *k*th occurrence (default, first) on each line of the given range. If **g** follows, substitute every occurrence on each line. If **p** follows, print the resulting current line.

[*n*[,*m*]]**t**[*d*]
Transfer (copy) selected lines to follow line *d* (default, current line).

[*n*]**u**[**p**]
Undo effect of last substitute command. If optional **p** specified, print undone line. The specified line must be the last substituted line.

[*n*[,*m*]]**v**/[*pattern*]/*commands*
Globally execute *commands* for each line in the specified range (default: all lines) *not* containing the *pattern* (default: current search pattern). The *commands* may extend over several lines, with all but the last terminated by '\'. The **v** command is like the **g** command, except the sense of the search is reversed.

[*n*[,*m*]]**w** [*file*]
Write selected lines (default, all lines) to *file* (default, current file name). The previous contents of *file*, if any, are lost.

[*n*[,*m*]]**W** [*file*]
Write specified lines (default, all lines) to the end of *file* (default, current file name). Like **w**, but appends to *file* instead of truncating it.

Patterns

Substitution commands and search specifications may include *patterns*, also called *regular expressions*. A non-special character in a pattern matches itself. Special characters include the following.

^ Match beginning of line, unless it appears immediately after '[' (see below).

$ Match end of line.

***** Matches zero or more repetitions of preceding character.

. Matches any character except newline.

[*chars*]
Matches any one of the enclosed *chars*. Ranges of letters or digits may be indicated using '-'.

[**^chars**]
Matches any character *except* one of the enclosed *chars*. Ranges of letters or digits may be indicated using '-'.

\c Disregard special meaning of character *c*.

\(**pattern**\)
Delimit substring *pattern* for use with **d**, described below.

The replacement part *pattern2* of the substitute command may also use the following:

& Insert characters matched by *pattern1*.

d Insert substring delimited by *d*th occurrence of delimiters '\(' and '\)', where *d* is a digit.

Options

The user may specify **ed** options on the command line, in the environment, or with the **o** command. The available options are as follows:

c Print character counts on **e**, **r**, and **w** commands.

m Allow multiple commands per line.

o Print line counts instead of character counts on **e**, **r**, and **w** commands.

p Prompt with an '*' for each command.

s Match lower-case letters in a *pattern* to both upper-case and lower-case text characters.

v Print verbose versions of error messages.

The **c** option is normally set, and all others are normally reset. Options may be set on the command line with a leading '+' sign. The '-' command line option resets the **c** option.

Options may be set in the environment with an assignment, such as

```
export ED=+cv
```

Options may be set with the '+' prefix or reset with the '-' prefix.

See Also

commands, elvis, ex, me, sed, vi
Introduction to ed

Diagnostics

ed usually prints only the diagnostic '?' on any error. When the verbose option **v** is specified, the '?' is followed by a brief description of the nature of the error.

EDITOR — Environmental Variable

Name editor to use by default
EDITOR=*editor*

The environmental variable **EDITOR** names the default editor that you wish to use. For example, **mail** invokes *editor* when you conclude a mail message by typing a question mark '?' at the beginning of a line followed by **<return>**. The screen pager **more** invokes *editor* when you enter the command **v** while displaying a file.

See Also

environmental variables, mail, more

egrep — Command

Extended pattern search
egrep [*option* ...] [*pattern*] [*file* ...]

egrep is an extended and faster version of **grep**. It searches each *file* for occurrences of *pattern* (also called a regular expression). If no *file* is specified, it searches the standard input. Normally, it prints each line matching the *pattern*.

Wildcards

The simplest *patterns* accepted by **egrep** are ordinary alphanumeric strings. Like **ed**, **egrep** can also process *patterns* that include the following wildcard characters:

^ Match beginning of line, unless it appears immediately after '[' (see below).

$ Match end of line.

* Match zero or more repetitions of preceding character.

. Match any character except newline.

[*chars*]
Match any one of the enclosed *chars*. Ranges of letters or digits may be indicated using '-'.

[^*chars*]
Match any character *except* one of the enclosed *chars*. Ranges of letters or digits may be indicated using '-'.

c Disregard special meaning of character *c*.

Metacharacters

In addition, **egrep** accepts the following additional metacharacters:

| Match the preceding pattern *or* the following pattern. For example, the pattern **cat|dog** matches either **cat** or **dog**. A newline within the *pattern* has the same meaning as '|'.

+ Match one or more occurrences of the immediately preceding pattern element; it works like '*', except it matches at least one occurrence instead of zero or more occurrences.

? Match zero or one occurrence of the preceding element of the pattern.

(...) Parentheses may be used to group patterns. For example, **(Ivan)+** matches a sequence of one or more occurrences of the four letters 'I' 'v' 'a' or 'n'.

Because the metacharacters '*', '?', '$', '(', ')', '[', ']', and '|' are also special to the shell **sh**, patterns that contain those literal characters must be quoted by enclosing *pattern* within single quotation marks.

Options

The following lists the available options:

-b With each output line, print the block number in which the line started (used to search file systems).

-c Print how many lines match, rather than the lines themselves.

-e The next argument is *pattern* (useful if the pattern starts with '-').

-f The next argument is a file that contains a list of patterns separated by newlines; there is no *pattern* argument.

-h When more than one *file* is specified, output lines are normally accompanied by the file name; **-h** suppresses this.

-l Print the name of each file that contains the string, rather than the lines themselves. This is useful when you are constructing a batch file.

-n When a line is printed, also print its number within the file.

-s Suppress all output, just return exit status.

-v Print a line only if the pattern is *not* found in the line.

-y Lower-case letters in the pattern match lower-case *and* upper-case letters on the input lines. A letter escaped with '\' in the pattern must be matched in exactly that case.

See Also

awk, cgrep, commands, ed, expr, grep, lex, sed

Diagnostics

egrep returns an exit status of zero for success, one for no matches, and two for error.

Notes

For matching patterns in C programs, the command **cgrep** is preferred, because it is optimized to recognize C-style expressions.

Besides the difference in the range of patterns allowed, **egrep** uses a deterministic finite automaton (DFA) for the search. It builds the DFA dynamically, so it begins doing useful work immediately. This means that **egrep** is much faster than **grep**, often by more than an order of magnitude, and is considerably faster than earlier pattern-searching commands, on almost any length of file.

else — C Keyword

Introduce a conditional statement

else is the flip side of an **if** statement: if the condition described in the **if** statement fails, then the statements introduced by the **else** statement are executed. For example,

```
if (getchar() == EOF)
     exit(0);
else
     dosomething();
```

exits if the user types **EOF**, but does something if the user types anything else.

See Also

C keywords, if

elvis — Command

Clone of Berkeley-standard screen editor
elvis [*options*] [*+cmd*] [*file1 ... file27*]

elvis is a clone of **vi** and **ex**, the standard UNIX screen editors.

Unlike MicroEMACS, the COHERENT system's other screen editor, **elvis** is a modal editor whose command structure resembles the **ed** line editor. *Modal* means that a keystroke assumes a different meaning, depending upon the mode that the editor is in. **elvis** uses three modes: *visual-command mode*, *colon-command mode*, and *input mode*. The following sections summarize the commands associated with each mode.

Visual-Command Mode

Visual-command mode closely resembles text-input mode. One quick way to tell the modes apart is to press the **<esc>** key. If **elvis** beeps, then you are in visual-command mode. If it does not beep, then you were in input mode, but pressing **<esc>** switched you to visual-command mode.

Most visual-mode commands are one keystroke long. The commands are in two groups: movement commands and edit commands. The former group moves the cursor through the file being edited, and the latter group alters text.

The following sections summarize the command set for **elvis**'s visual-command mode.

Visual-Mode Movement Commands

The following summarizes the visual mode's movement commands. *count* indicates that the command can be optionally prefaced by an argument that tells **elvis** how often to execute the command. *move* indicates that the command can be followed by a movement command, after which the command is executed on the text that lies between the point where the command was first typed and the point to which the cursor was moved. Typing the command a second time executes the command for the entire line upon which the cursor is positioned. *key* means that the command must be followed by an argument. The following describes

<ctrl-B>	Move up by one screenful.
[count] **<ctrl-D>**	Scroll down *count* lines (default, one-half screenful).
[count] **<ctrl-E>**	Scroll up *count* lines.
<ctrl-F>	Move down by one screenful.
<ctrl-G>	Show file status and the current line line.
[count] **<ctrl-H>**	Move one character to the left.
[count] **<ctrl-J>**	Move down *count* lines.
<ctrl-L>	Redraw the screen.
[count] **<ctrl-M>**	Move to the beginning of the next line.
[count] **<ctrl-N>**	Move down *count* lines (default, one).
[count] **<ctrl-P>**	Move up *count* lines (default, one).
<ctrl-R>	Redraw the screen.
[count] **<ctrl-U>**	Scroll up *count* lines (default, one-half screenful).
[count] **<ctrl-Y>**	Scroll down *count* lines.
<ctrl-]>	If the cursor is on a tag name, go to that tag.
<ctrl-^>	Switch to the previous file.

[count] **<space>**	Move right *count* spaces (default, one).
! *[move]*	Run the selected text through an external filter program.
" *key*	Select which cut buffer to use next.
$	Move to the end of the current line.
%	Move to the matching (){}[] character.
' *key*	Move to a marked line.
[count] **(**	Move backward *count* sentences (default, one).
[count] **)**	Move forward *count* sentences (default, one).
*****	Go to the next error in the error list.
[count] **+**	Move to the beginning of the next line.
[count] **,**	Repeat the previous **f** or **t** command, but move in the opposite direction.
[count] **—**	Move to the beginning of the preceding line.
[count] **.**	Repeat the previous *edit* command.
/ *text*	Search forward for *text*, which can be a regular expression.
0	If not part of a count, move to the first character of this line.
:	Switch to colon-command mode to execute one command.
[count] **;**	Repeat the previous **f** or **t** command.
? *text*	Search backwards for *text*, which can be a regular expression.
@ *key*	Execute the contents of a cut-buffer as **vi** commands.
[count] **B**	Move backwards *count* words (default, one).
[count] **E**	Move forwards *count* words (default, one).
[count] **F** *key*	Move left to the *count*'th occurrence of the given character (default, first).
[count] **G**	Move to to the *count*'th line in the file (default, last).
[count] **H**	Move to the top of the screen.
K	Look up a keyword.
[count] **L**	Move to the bottom of the screen.
M	Move to the middle of the screen.
N	Repeat the last search, but in the opposite direction.
P	Paste text before the cursor.
Q	Shift to colon-command mode.
[count] **T** *key*	Move left *almost* to the given character.
U	Undo all recent changes to the current line.

LEXICON

[count] **U**	Move forward *count* words (default, one).
[count] **Y**	Copy (or "yank") *count* lines into a cut buffer (default, one).
Z Z	Save the file and exit.
[[	Move back one section.
]]	Move forward one section.
^	Move to the beginning of the current line, but after indent.
' *key*	Move to the *key* character.
[count] **b**	Move back *count* words.
[count] **e**	Move forward to the end of the *count*'th word.
[count] **f** *key*	Move rightward to the *count*'th occurrence of the given character.
[count] **h**	Move left *count* characters (default, one).
[count] **j**	Move down *count* characters (default, one).
[count] **k**	Move up *count* characters (default, one).
[count] **l**	Move right *count* characters (default, one).
m *key*	Mark a line or character.
n	Repeat the previous search.
p	Paste text after the cursor.
[count] **t** *key*	Move rightward *almost* to the *count*'th occurrence of the given character (default, one).
u	Undo the previous edit command.
[count] **w**	Move forward *count* words (default, one).
y *move*	Copy (or "yank") text into a cut buffer.
z *key*	Scroll the screen, repositioning the current line as follows: **+** indicates top of the screen, **—** indicates the bottom, **.** indicates the middle.
[count] **{**	Move back *count* paragraphs (default, one).
[count] **|**	Move to the *count*'th column on the screen (leftmost, one).
[count] **}**	Move forward *count* paragraphs (default, one).

Visual-Mode Edit Commands

The following describes the visual mode's editing commands.

[count] **#**	Increment a number by *count* (default, one).
[count] **&**	Repeat the previous **:s//** command.
< *move*	Shift the enclosed text left.
> *move*	Shift the enclosed text right.
[count] **A** *input*	Append input at end of the line.

C *input*	Change text from the cursor through the end of the line.
D	Delete text from the cursor through the end of the line.
[count] **I** *input*	Insert text at the beginning of the line (after indentations).
[count] **J**	Join lines the current with the following line.
[count] **O** *input*	Open a new line above the current line.
R *input*	Overtype.
[count] **S** input	Change lines, like **cc**.
[count] **X**	Delete *count* characters from the left of the cursor (default, one).
[count] **a** *input*	Insert text after the cursor.
c *move*	Change text.
d *move*	Delete text.
[count] **i** *input*	Insert text at the cursor.
[count] **o** *input*	Open a new line below the current line.
[count] **r** *key*	Replace *count* characters with text you type (default, one).
[count] **s** *input*	Replace *count* characters with text you type (default, one).
[count] **x**	Delete the character at which the cursor is positioned.
[count] ~	Toggle a character between upper case and lower case.

Colon-Mode Commands

The following summarizes the set of colon-mode commands. It is no accident that these commands closely resemble those for the **ed** line editor: they come, in fact, from **ex**, the editor upon which both **vi** (the UNIX visual editor) and **ed** derive. For that reason, colon-command mode is sometimes called **ex** mode.

line indicates whether the command can be executed on one or more lines. *line* can be a regular expression. Some commands can be used with an optional exclamation point; if done so, the editor assumes you know what you are doing and suppresses the warnings and prompts it would normally issue for these commands. Please note, finally, that most commands can be invoked simply by typing the first one or two letters of their names.

abbr *[word full_form]*	Define *word* as an abbreviation for *full_form*.
[line] **append**	Insert text after the current line.
args *[file1 ... fileN]*	With no arguments, print the files list on **elvis**'s command line. With one or more arguments, change the name of the current file.
cc *[files]*	Invoke the C compiler to compile *files*, and redirects all error messages into file **errlist**. After the compiler exits, scan the contents of **errlist** for error messages; if one is found, jump to the line and file indicated on the error line, and display the error message on the status line.
cd *[directory]*	Switch the current working directory. With no argument, switch to the **$HOME** directory.

[line][,line] **change** *["x]*
Replace the range of lines with the contents of cut-buffer *x*.

chdir *[directory]* Same as the **cd** command.

[line][,line] **copy** *targetline*
Copy the range of lines to after the *targetline*.

[line][,line] **delete** *["x]*
Move the range of lines into cut buffer *x*.

digraph*[!] [XX [Y]]* Set *XX* as a digraph for *Y*. With no arguments, display all currently defined digraphs. With one argument, undefine the argument as a digraph.

edit*[!] [file]* Edit a file not named on the **elvis** command line.

errlist*[!] [errlist]* Find the next error message in file **errlist**, as generated through **elvis**'s **cc** or **make** commands.

file *[file]* With an argument, change the output file to *file*. Without an argument, print information about the current output file.

[line][,line] **global** */regexp/ command*
Search the range of lines for all lines that contain the regular expression *regexp*, and execute *command* upon each.

[line] **insert** Insert text before the current line.

[line][,line] **join** Concatenate the range of lines into one line.

[line][,line] **list** Display the requested range of lines, making all embedded control characters explicit.

make *[target]* Same as the **cc** command, except that **make** is executed.

map*[!] key mapped_to*
Remap *key* to *mapped_to*. Normally, remapping applies just to visual-command mode; '!' tells **elvis** to remap the key under all modes. With no arguments, show all current key mappings.

[line] **mark** *x* Set a mark on *line*, and name it *x*.

mkexrc Save current configuration into file **./.exrc**, which will be read next time you invoke **elvis**.

[line][,line] **move** *targetline*
Move the range of lines to after *targetline*.

next*[!] [files]* Switch to the next file on the **elvis** command line.

Next*[!]* Switch to the preceeding file on the **elvis** command line.

[line][,line] **number** Display the range of lines, with line numbers.

previous*[!]* Switch to the preceeding file on the **elvis** command line.

[line][,line] **print** Display the specified range of lines.

[line] **put** *["x]* Copy text from cut buffer *x* after the current line.

quit*[!]* Quit **elvis**, and return to the shell.

[line] **read** *file* — Read the contents of *file* and insert them after **line** (default, the last line).

rewind*[!]* — Switch to the first file on the **elvis** command line.

set *[options]* — Set an **elvis** option. With no arguments, list current settings for all options.

shell — Invoke a shell.

source *file* — Read a set of colon-mode commands from *file*, and execute them.

[line][,line] **substitute** */regexp/replacement/[p][g][c]*
For the range of lines, replace the first instance of *regexp* with *replacement*. *p* tells **elvis** to print the *last* line upon which a substitution was performed. *g* means perform a global substitution, i.e., replace all instances of *regexp* on each line with *replacement*. *c* tells **elvis** to ask for confirmation before performing each substitution.

tag*[!] tagname* — Find *tagname* in file **tags**, which records information about all tags. If found, jump to the file and line upon which the tag is set.

[line][,line] **to** *targetline*
Copy the range of lines to after the *targetline*.

unabbr *word* — Unabbreviate *word*.

undo — Undo the last editing command.

unmap*[!] key* — Unmap *key*.

version — Display the current version of **elvis**.

[line][,line] **vglobal** */regexp/ command*
Search the range of lines for all lines that do not contain the regular expression *regexp*, and execute *command* upon each.

visual — Enter visual-command mode.

wq — Save the changed file, and exit.

[line][,line] **write***[!] [[>>]file]*
Write the file being edited into *file*. With the >> argument, append the edited text onto the end of *file*.

xit*[!]* — Same as the **wq** command, described above, except that it does not write files that have not changed.

[line][,line] **yank** *["x]* — Copy the range of lines into cut buffer *x*.

[line][,line] **!** *command*
Execute *command* under a subshell, then return.

[line][,line] **<** — Shift the range of lines left by one tabwidth.

[line][,line] **=** — With no range of lines specified, print the number of the current line. With line arguments, print the endpoints of the lines in question, and the number of lines that lie between them. (Remember, *line* can be a regular expression as well as a number.)

[line][,line] **>** — Shift the range of lines right by one tabwidth.

[line][,line] **&**	Repeat the last substitution command.
@ x	Read the contents of cut-buffer *x* as a set of colon-mode commands, and execute them.

Input-Mode Commands

Most keystrokes are interpreted as being text and inserted directly into the text; however, some keystrokes are still interpreted as commands. Thus, you can perform an entire session of simple editing directly within input mode without switching to either of the command modes.

The following summarizes the commands that can be executed directly within input mode:

<ctrl-A>	Insert a copy of the last input text.
<ctrl-D>	Delete one indent character.
<ctrl-H>	Erase the character before the cursor.
<ctrl-L>	Redraw the screen.
<ctrl-M>	Insert a newline.
<ctrl-P>	Insert the contents of the cut buffer.
<ctrl-R>	Redraw the screen, like **<ctrl-L>**.
<ctrl-T>	Insert an indent character.
<ctrl-U>	Move to the beginning of the line.
<ctrl-V>	Insert the following keystroke, even if special.
<ctrl-W>	Backspace to the beginning of the current word.
<ctrl-Z><ctrl-Z>	Write the file and exit **elvis**.
<esc>	Shift from input mode to visual-command mode.
<del>	Delete the current character.

Command-line Options

elvis lets you name up to 27 files on the command line, thus allowing you to edit up to 27 files simultaneously. The "next file" and "previous file" commands described above allow you to shift from one file to another during the same editing session; in this way, for example, you can cut text from one file and paste it into another.

elvis recognizes the following command-line options:

-r Recover a previous edit. Because **elvis** uses the program **virec** for file recovery, invoking it with this option simply prints a message that tells you to run **virec**.

-R Invoke **elvis** in "read-only" mode. This is equivalent to invoking **elvis** via the link **view**.

-t *tag* Begin editing at *tag*.

-m *[file]*
Invoke **elvis** in error-handling mode. It searches through *file* for something that looks like an error message from a compiler, then positions the cursor at that point for editing.

-e Begin in colon-command mode.

-v Begin in visual-command mode.

-i Begin in input mode.

+*command*
Execute *command* immediately upon beginning editing. For example

```
elvis +237 foo
```

causes **elvis** to move directly to line 237 immediately upon beginning to edit file **foo**.

Files

/tmp/elv* — Temporary files

See Also

commands, ed, ex, me, vi, view

Notes

Full documentation for **elvis** is included with this release in compressed file **/usr/src/alien/Elvis.doc.Z**.

elvis is copyright © 1990 by Steve Kirkendall, and was written by Steve Kirkendall (kirkenda@cs.pdx.edu or ...uunet!tektronix!psueea!eecs!kirkenda), assisted by numerous volunteers. It is freely redistributable, subject to the restrictions noted in included documentation. Source code for **elvis** is available through the Mark Williams bulletin board, USENET, and numerous other outlets.

elvis is distributed as a service to COHERENT customers, as is. It is not supported by Mark Williams Company. *Caveat utilitor.*

enable — Command

Enable a port
/etc/enable *port...*

The COHERENT system is a multiuser operating system; it allows many users to use the system simultaneously. An asynchronous communication *port* connects each user to the system, normally by a terminal or a modem attached to the port. The system communicates with the port by means of a character special file in directory **/dev**, such as **/dev/com3r** or **/dev/com2l**.

The COHERENT system will not allow a user to log in on a port until the system creates a *login process* for the port. The **enable** command tells the system to create a login process for each given *port*. For example, the command

```
/etc/enable com1r
```

enables port **/dev/com1r**.

enable changes the entry for each given *port* in the terminal characteristics file **/etc/ttys**. The baud rate specified in **/etc/ttys** must be the appropriate baud rate for the terminal or modem connected to the port. See the Lexicon entry for **ttys** for more information.

The command **disable** disables a port. The command **ttystat** checks whether a port is enabled or disabled.

Files

/etc/ttys — Terminal characteristics file
/dev/com* — Devices serial ports

See Also

com, commands, disable, getty, login, ttys, ttystat

Diagnostics

enable normally returns one if it enables the *port* successfully and zero if not. If more than one *port* is specified, **enable** returns the success or failure status of the last port it finds. It returns -1 if it cannot find any given *port*. An exit status of -2 indicates an error.

Notes

It is not recommended that you attempt to enable a port that is already enabled. To make sure, run **/etc/disable** before running **/etc/enable**.

Only the superuser **root** can execute **enable**.

end — Linker-Defined Symbol

extern int end[];
end is the location after the uninitialized data segment. It is defined by the linker when it binds the program together for execution. The value of **end** is merely an address. The location to which it points contains no known value, and may be illegal memory locations for the program. The value of **end** does not change while the program is running.

Example

```
main()
{
     extern int end[];
     printf("end=%lx\n", (long) end);
}
```

See Also

etext, ld, linker-defined symbols, malloc()

endgrent() — General Function (libc)

Close group file
#include <grp.h>
endgrent()

endgrent closes the file **/etc/group**. It returns NULL if an error occurs.

Files

/etc/group
<grp.h>

See Also

general functions, group

endpwent() — General Function (libc)

Close password file
#include <pwd.h>
endpwent()

The COHERENT system has five routines that search the file **/etc/passwd**, which contains information about every user of the system. **endpwent** closes the password file.

Files

/etc/passwd
<pwd.h>

See Also

general functions, getpwent(), getpwnam(), getpwuid(), pwd.h, setpwent()

Diagnostics

endpwent returns NULL for any error or on end of file.

enum — C Keyword

Declare a type and identifiers

An **enum** declaration is a data type whose syntax resembles those of the **struct** and **union** declarations. It lets you enumerate the legal value for a given variable. For example,

```
enum opinion {yes, maybe, no} GUESS;
```

declares type **opinion** can have one of three values: **yes**, **no**, and **maybe**. It also declares the variable **GUESS** to be of type **opinion**.

As with a **struct** or **union** declaration, the tag (**opinion** in this example) is optional; if present, it may be used in subsequent declarations. For example, the statement

```
register enum opinion *op;
```

declares a register pointer to an object of type **opinion**.

All enumerated identifiers must be distinct from all other identifiers in the program. The identifiers act as constants and be used wherever constants are appropriate.

COHERENT assigns values to the identifiers from left to right, normally beginning with zero and increasing by one. In the above example, the values of **yes**, **no**, and **maybe** are set, respectively, to one, two, and three. The values often are **ints**, although if the range of values is small enough, the **enum** will be an **unsigned char**. If an identifier in the declaration is followed by an equal sign and a constant, the identifier is assigned the given value, and subsequent values increase by one from that value; for example,

```
enum opinion {yes=50, no, maybe} guess;
```

sets the values of the identifiers **yes**, **no**, and **maybe** to 50, 51, and 52, respectively.

See Also

C keywords

ENV — Environmental Variable

File read to set environment

The Korn shell reads the environmental variable **ENV** to determine what file to read after it executes the contents of **profile** and **.profile**. This file is usually used to set aliases and variables.

See Also

environmental variables, ksh

environ — Technical Information

Process environment

extern char **environ;

environ is an array of strings, called the *environment* of a process. By convention, each string has the form

name=value

Normally, each process inherits the environment of its parent process. The shell **sh** and various forms of **exec** can change the environment. The shell adds the name and value of each shell variable marked for *export* to the environment of subsequent commands. The shell adds assignments given on the same line as a command to the environment of the command, without affecting subsequent commands.

See Also

exec, getenv(), sh, technical information

environmental variables — Overview

The *environment* is a set of information that is read by all programs that run on your system. It consists of one or more *environmental variables* that you set. For example, when you set the environmental variable **PATH**, you tell COHERENT that you wish to pass this information to all programs on your system, including COHERENT itself.

By changing the environment, you can change the way a command works without rewriting any commands that you may have embedded in batch files, scripts, or **makefiles**.

COHERENT uses the following environmental variables to set is environment. The programs that you write may use others that you define yourself, if you wish.

ASKCC	Have **mail** prompt for CC names
CWD	Current working directory
EDITOR	Editor used by default by **mail**
ENV	File read to set environment
FCEDIT	Editor used by the **fc** command
IFS	Characters recognized as white space
HOME	User's home directory
KSH_VERSION	List current version of Korn shell
LASTERROR	Program that last generated an error
MAIL	File that holds user's mail messages
PAGER	User's preferred output filter
PATH	Directories that hold executable files
PS1	User's default prompt
PS2	Prompt when unbalanced quotation marks span a line
SECONDS	Number of seconds since current shell started
SHELL	Name the default shell
TERM	Name the default terminal type
TIMEZONE	User's current time zone
USER	Name user's identifier

See Also

Lexicon

envp — C Language

Argument passed to main()
char *envp[];

envp is an abbreviation for environmental parameter. It is the traditional name for a pointer to an array of string pointers passed to a C program's **main** function, and is by convention the third argument passed to **main**.

Example

The following example demonstrates **envp**, **argc**, and **argv**.

```
#include <stdio.h>

main(argc, argv, envp)
int argc;                        /* Number of args */
char *argv[];                    /* Argument ptr array */
char *envp[];                    /* Environment ptr array */
{
     int a;

     printf("The command name (argv[0]) is %s\n", argv[0]);
     printf("There are %d arguments:\n", argc-1);
     for (a=1; a<argc; a++)
          printf("\targument %2d:\t%s\n", a, argv[a]);

     printf("The environment is as follows:\n");
     a = 0;
     while (envp[a] != NULL)
          printf("\t%s\n", envp[a++]);
}
```

See Also

argc, argv, C language, environ, main()

EOF — Definition (Library/STDIO)

Indicate end of a file
#include <stdio.h>

EOF is an indicator that is returned by several STDIO functions to indicate that the current file position is the end of the file.

Many STDIO functions, when they read **EOF**, set the end-of-file indicator that is associated with the stream being read. Before more data can be read from the stream, its end-of-file indicator must be cleared. Resetting the file-position indicator with the functions **fseek**, **fsetpos**, or **ftell** will clear the indicator, as will returning a character to the stream with the function **ungetc**.

See Also

file, stream, STDIO, stdio.h

epson — Command

Print files on Epson printer
epson [**-cdfrw8**] [**-b** *head*] [**-i** *n*] [**-o** *ofile*] [**-s** *n*] [*file* ...]

epson prints each *file*, or the standard input if none, on an Epson MX-80 printer or compatible. **epson** normally sends its output directly to the line printer **/dev/lp**. It recognizes the **nroff** output sequences for boldface and italics and normally converts them to emphasized print and italics.

epson recognizes the following options:

-b *head*
: Print the given *head* as a double-width banner at the top of the first output page.

-c Use compressed printing mode.

-d Print boldface as double strikes. Normally, **epson** recognizes the sequence "*c***b***c*" as boldface and prints *c* in emphasized printing mode. **-d** is useful in conjunction with **-c**.

-f Do not print a formfeed character at the end of each *file*.

-i*n* Indent *n* spaces at the start of each output line.

-o *ofile*
: Send output to *ofile* instead of **/dev/lp**.

-r Print all characters in Roman; do not use italics. Normally, **epson** recognizes the sequence "_**b***c*" as italic and prints *c* in its italic character set.

-s*n* Print *n* newlines at the end of each line. *n* must be 1, 2, or 3; the default is 1.

-w Use double width printing mode.

-8 Print lines with vertical spacing of eight lines per inch instead of the default six lines per inch.

Files

/dev/lp — Line printer

See Also

commands, lpr, nroff, pr, printer

Diagnostics

epson prints appropriate messages on the standard error if it cannot open a *file* or if an argument is incorrect.

errno — Technical Information

External integer for return of error status
extern int errno;

errno is an external integer that COHERENT links into every program. COHERENT sets **errno** to the negative value of any error status returned by COHERENT to the functions that perform COHERENT system calls.

Mathematical functions use **errno** to indicate classifications of errors on return. **errno** is defined within the header file **errno.h**. Because not every function uses **errno**, it should be polled only in connection with those functions that document its use and the meaning of the various status values. For the names of the error codes (as defined in **errno.h**, their

value, and the message returned by the function **perror**, see **errno.h**.

Example

For an example of using **errno** in a mathematics program, see the entry for **acos**.

See Also

errno.h, mathematics library, perror(), signal(), technical information

errno.h — Header File

Error numbers used by errno()
#include <errno.h>

errno.h is a header that defines and describes the error numbers returned in the external variable **errno**. The following lists the the error numbers defined in **errno.h**:

EPERM: Not super user
You are not the superuser **root**, and attempted an operation that requires **root** privileges.

ENOENT: No such file or directory
A program could not find a required file or directory.

ESRCH: Process not found
A program attempt to communicate with a process that did not exist.

EINTR: Interrupted system call
A COHERENT system call failed due to a signal being received or an alarm expiring.

EIO: I/O error
A physical I/O error occurred on a device driver. This could be a tape error, a CRC error on a disk, or a framing error on a synchronous HDLC link.

ENXIO: no such device or address
A specified minor device is invalid or the unit is powered off. This error might also indicate that a block number given to a minor device is out of range. **suload** returns this error code if the driver was not loaded.

E2BIG: argument list too long
The number of bytes of arguments passed in an **exec** is too large.

ENOEXEC: exec format error
The file given to **exec** or **load** is not a valid load module (probably because it does not have the magic number at the beginning), even though its mode indicates that it is executable.

EBADF: bad file descriptor
A file descriptor passed to a system call is not open or is inappropriate to the call. For example, a file descriptor opened only for reading may not be accessed for writing.

ECHILD: no children
A process issued a **wait** call when it had no outstanding children.

EAGAIN: no more processes
The system cannot create any more processes, either because it is out of table space or because the invoking process has reached its process quota.

ENOMEM: not enough memory
The system cannot accomodate the memory size requested (by **exec** or **brk**, for example).

EACCES: permission denied
The user is denied access to a file.

EFAULT: bad address
An address in a system call does not lie in the address space. Normally, this generates a **SIGSYS** signal, which terminates the process.

ENOTBLK: block device required
The **mount** and **umount** calls require block devices as arguments.

EBUSY: mount device busy
The special file passed to **mount** is already mounted, or the file system given to **umount** has open files or active working directories.

EEXIST: file exists
An attempt was made to **link** to a file that already exists.

EXDEV: cross-device link
A link to a file must be on the same logical device as the file.

ENODEV: no such device
An unsuitable I/O call was made to a device; for example, an attempts to read a line printer.

ENOTDIR: not a directory
A component in a path name exists but is not a directory, or a **chdir** or **chroot** argument is not a directory.

EISDIR: is a directory
Directories cannot be opened for writing.

EINVAL: invalid argument
An argument to a system call is out of range, e.g., a bad signal number to **kill** or **umount** of a device that is not mounted.

ENFILE: file table overflow
A table inside the COHERENT system has run out of space, preventing further **open** calls and related requests.

EMFILE: too many open files
A process is limited to 20 open files at any time.

ENOTTY: not a tty
An **ioctl** call was made to a file which is not a terminal device.

ETXTBSY: text file busy
The text segment of a shared load module is unwritable. Therefore, an attempt to execute it while it is being written or an attempt to open it for writing while it is being executed will fail.

EFBIG: file too large
The block mapping algorithm for files fails above 1,082,201,088 bytes.

ENOSPC: no space left on device
Indicates an attempt to write on a file when no free blocks remain on the associated device. This error may also indicate that a device is out of i-nodes, so a file cannot be created.

ESPIPE: illegal seek
It is illegal to **lseek** on a pipe.

EROFS: read-only file system
Indicates an attempt to write on a file system mounted read-only (e.g., with **creat** or **unlink**).

EMLINK: too many links
A new link to a file cannot be created, because the link count would exceed 32,767.

EPIPE: broken pipe
A write occurred on a pipe for which there are no readers. This condition is accompanied by the signal **SIGPIPE**, so the error will only be seen if the signal is ignored or caught.

EDOM: mathematics library domain error
An argument to a mathematical routine falls outside the domain of the function.

ERANGE: mathematics library result too large
The result of a mathematical function is too large to be represented.

EKSPACE: out of kernel space
No more space is available for tables inside the COHERENT system. Table space is dynamically allocated from a fixed area of memory; it may be possible to increase the size of the area by reconfiguring the system.

ENOLOAD: driver not loaded
Not used.

EBADFMT: bad exec format
An attempt was made to **exec** a file on the wrong type of processor.

EDATTN: device needs attention
The device being referenced needs operator attention. For example, a line printer might need paper.

EDBUSY: device busy
The indicated device is busy. For **load**, this implies that the given major device number is already in use.

See Also

errno, header files, perror(), signal()

etext — Linker-Defined Symbol

extern int etext[];

etext is the location after the shared and private text (code) segments. It is defined by the linker when it binds the program together for execution. The value of **etext** is merely an address. The location to which it points contains no known value, and may be illegal memory locations for the program. The value of **etext** does not change while the program is running.

Example

```
main()
{
	extern int etext[];
	printf("etext=%ld\n", (long) etext);
}
```

See Also

brk(), end, ld, linker-defined symbols, malloc()

eval — Command

Evaluate arguments
eval [*token* ...]

The shell normally evaluates each token of an input line before executing it. During evaluation, the shell performs parameter, command, and file-name pattern substitution. The shell does *not* interpret special characters after performing substitution.

eval is useful when an additional level of evaluation is required. It evaluates its arguments and treats the result as shell input. For example,

```
A='>file'
echo a b c $A
```

simply prints the output

```
a b c >file
```

because '>' has no special meaning after substitution, but

```
A='>file'
eval echo a b c $A
```

redirects the output

```
a b c
```

to **file**. Similarly,

```
A='$B'
B='string'
echo $A
eval echo $A
```

prints

```
$B
string
```

In the first **echo** the shell performs substitution only once.

The shell executes **eval** directly.

See Also

commands, ksh, sh

ex — Command

Berkeley-style line editor
ex [*options*] [*+cmd*] [*file1* ... *file27*]

ex is a link to **elvis**, which is a clone of the UNIX **vi/ex** set of editors. Invoking **elvis** through this link forces it to operate solely in colon-command mode, just as the UNIX **ex** editor operates.

For information on how to use this version of **ex**, see the Lexicon page for **elvis**.

See Also

commands, ed, elvis, me, vi, view

Notes

elvis is copyright © 1990 by Steve Kirkendall, and was written by Steve Kirkendall (kirkenda@cs.pdx.edu or ...uunet!tektronix!psueea!eecs!kirkenda), assisted by numerous volunteers. It is freely redistributable, subject to the restrictions noted in included documentation. Source code for **elvis** is available through the Mark Williams bulletin board, USENET, and numerous other outlets.

Please note that **elvis** is distributed as a service to COHERENT customers, as is. It is not supported by Mark Williams Company. *Caveat utilitor.*

exec — Command

Execute command directly
exec [*command*]

The shell normally executes commands with a **fork** system call, which creates a new process. The shell command **exec** directly executes the given *command* with an **exec** system call instead. Normally, this terminates execution of the current shell.

If the *command* consists only of redirection specifications, **exec** redirects the input or output of the current shell accordingly without terminating it. If the *command* is omitted, **exec** has no effect.

See Also

commands, fork(), ksh, sh

execl() — COHERENT System Call (libc)

Execute a load module
execl(*file, arg0, arg1, ..., argn,* **NULL**)
char **file, *arg0, *arg1, ..., *argn;*

The COHERENT system includes six system calls that allow a process to execute another executable *file*. **execl** specifies arguments individually, as a NULL-terminated list of *arg* parameters. For more information on file execution, see **execution**.

See Also

COHERENT system calls, execution, getuid()

Diagnostics

execl does not return if successful. It returns -1 for errors, such as *file* being nonexistent, not accessible with execute permission, having a bad format, or too large to fit in memory.

execle() — COHERENT System Call (libc)

Execute a load module
execle(*file, arg0, arg1, ..., argn,* **NULL**, *env*)
char **file, *arg0, *arg1, ..., *argn*, **char** **env*[];

The COHERENT system includes six functions that allow a process to execute another executable *file* (or load module, as described in the header **l.out.h**). **execle** initializes the new stack of the process to contain a list of strings that are command arguments. It specifies arguments individually, as a NULL-terminated list of *arg* parameters. The argument *envp* points to an array of pointers to strings that define *file*'s environment. For more information on program execution and environments, see **execution**.

See Also

COHERENT system calls, environ, execution

Diagnostics

execle does not return if successful. It returns -1 for errors, such as *file* being nonexistent, not accessible with execute permission, having a bad format, or being too large to fit into memory.

execlp() — COHERENT System Call (libc)

Execute a load module
execlp(*file, arg0, arg1, ..., argn,* **NULL)**
char **file, *arg0, *arg1, ..., *argn;*

The COHERENT system includes six functions that allow a process to execute another executable *file*. **execlp** initializes the new stack of the process to contain a list of strings that are command arguments. It specifies arguments individually, as a NULL-terminated list of *arg* parameters. Unlike the related function **execl**, **execlp** searches for *file* in all directories named in the environmental variable **PATH**. For more information on program execution, see **execution**.

See Also

COHERENT system calls, environ, execution

Diagnostics

execlp does not return if successful. It returns -1 for errors, such as *file* not existing in the directories named in **PATH**, not accessible with execute permission, having a bad format, or too large to fit in memory.

executable file — Definition

An **executable file** is one that can be loaded directly by the operating system and executed. Normally, an executable file is one that has both been *compiled*, where it is rendered into machine language, and *linked*, where the compiled program has received all operating system-specific information and library functions.

See Also

definitions, file, object format

execution — Technical Information

Program execution under COHERENT is governed by the various forms of the COHERENT system call **exec**. This call allows a process to execute another executable *file* (load module, as described in **l.out.h**). The code, data and stack of *file* replace those of the requesting process. The new stack contains the command arguments and its environment, in the format given below. Execution starts at the entry point of *file*.

During a successful **exec**, the system deactivates profiling, and resets any caught signals to **SIG_DFL**.

Every process has a real-user id, an effective-user id, a real-group id, and an effective-group id, as described in **getuid**. For most load modules, **exec** does not change any of these. However, if the *file* is marked with the set user id or set group id bit (see **stat**), **exec** sets the effective-user id (effective-group id) of the process to the user id (group id) of the *file* owner. In effect, this changes the file access privilege level from that of the real id to that of the effective id. The owner of *file* should be careful to limit its abilities, to avoid compromising file security.

exec initializes the new stack of the process to contain a list of strings which are command

arguments. **execl**, **execle**, and **execlp** specify arguments individually, as a NULL-terminated list of *arg* parameters. **execv**, **execve**, and **execvp** specify arguments as a single NULL-terminated array *argv* of parameters.

The **main** routine of a C program is invoked in the following way:

```
main(argc, argv, envp)
int argc;
char *argv[], *envp[];
```

argc is the number of command arguments passed through **exec**, and **argv** is an array of the actual argument strings. **envp** is an array of strings that comprise the process environment. By convention, these strings are of the form *variable=value*, as described in the Lexicon entry **environ**. Typically, each *variable* is an **export**ed shell variable with the given *value*.

execl and **execv** simply pass the old environment, referenced by the external pointer **environ**. **execle** and **execve** pass a new environment *env* explicitly. **execlp** and **execvp** search for *file* in each of the directories indicated by the shell variable **$PATH**, in the same way that the shell searches for a command. These calls will execute a shell command *file*.

Files

/bin/sh — To execute command files

See Also

environ, execl(), execle(), execlp(), execv(), execve(), execvp(), fork(), ioctl(), signal(), stat(), technical information

Diagnostics

None of the **exec** routines returns if successful. Each returns -1 for errors, such as if *file* is nonexistent, not accessible with execute permission, has a bad format, or is too large to fit in memory.

execv() — COHERENT System Call (libc)

Execute a load module
execv(*file, argv***)**
char ****file*, ******argv*[];**

The COHERENT system includes six functions that allow a process to execute another executable *file* (or load module, as described in the header **l.out.h**). **execv** specifies arguments as a single, NULL-terminated array of parameters, called *argv*. Unlike the related program **execve**, **execv** passes the environment of the calling program to the called program. For more information on program execution, see **execution**.

See Also

COHERENT system calls, environ, execution

Diagnostics

execv does not return if successful. It returns -1 for errors, such as *file* being nonexistent, not accessible with execute permission, having a bad format, or too large to fit in memory.

execve() — COHERENT System Call (libc)

Execute a load module
execve(*file, argv, env***)**
char ****file*, ******argv*[], ******env***[];

The COHERENT system includes six functions that allow a process to execute another executable *file* (or load module, as described in the header **l.out.h**). **execve** specifies arguments as a single, NULL-terminated array of parameters, called *argv*. The argument *env* is the address of an array of pointers to strings that define *file*'s environment. This allows **execve** to pass a new environment to the program being executed. For more information on program execution, see **execl**.

Example

The following example demonstrates **execve**, as well as **tmpnam**, **getenv**, and **path**. It finds all lines with more than **LIMIT** characters and call MicroEMACS to edit them.

```
#include <stdio.h>
#include <path.h>
#include <sys/stat.h>

#define LIMIT 70

extern char *getenv(), **environ, *tempnam();

main(argc, argv)
char *argv[];
{
     /*                me     -e   tmp   file */
     char *cmda[5] = { NULL, "-e", NULL, NULL, NULL };
     FILE *ifp, *tmp;
     char line[256];
     int  ct, len;

     if ((NULL == (cmda[3] = argv[1])) ||
         (NULL == (ifp = fopen(argv[1], "r")))) {
          fprintf(stderr, "Cannot open %s\n", argv[1]);
          exit(1);
     }

     if (cmda[0] = path(getenv("PATH"), "me", AEXEC) == NULL) {
          fprintf(stderr, "Cannot locate me\n");
          exit(1);
     }

     if (NULL == (tmp = fopen((cmda[2] = tempnam(NULL, "lng")), "w"))) {
          fprintf(stderr, "Cannot open tmpfile\n");
          exit(1);
     }

     for (ct = 1; NULL != fgets(line, sizeof(line), ifp); ct++)
          if (((len = strlen(line)) > LIMIT) ||
              ('\n' != line[len -1]))
               fprintf(tmp, "%d: %d characters long\n", ct, len);

     fclose(tmp);
     fclose(ifp);
```

```
        if (execve(cmda[0], cmda, environ) < 0) {
                fprintf(stderr, "cannot execute me\n");
                exit(1);
        }
        /* We never reach here ! */
}
```

See Also

COHERENT system calls, environ, execution

Diagnostics

execve does not return if successful. It returns -1 for errors, such as *file* being nonexistent, not accessible with execute permission, having a bad format, or too large to fit in memory.

execvp() — COHERENT System Call (libc)

Execute a load module
execvp(*file, argv***)**
char ****file*, ******argv*[];**

The COHERENT system includes six functions that allow a process to execute another executable *file* (or load module, as described in the header **l.out.h**). **execvp** specifies arguments as a single, NULL-terminated array of parameters, called *argv*. Unlike the related call **execv**, **execvp** searches for *file* in all of the directories named in the environmental variable **PATH**. For more information on program execution, see **execution**.

See Also

COHERENT system calls, environ, execution

Diagnostics

execvp does not return if successful. It returns -1 for errors, such as *file* being nonexistent, not accessible with execute permission, having a bad format, or too large to fit in memory.

exit — Command

Exit from a noninteractive shell
exit [*status*]

exit terminates a noninteractive shell. If the optional *status* is specified, the shell returns it; otherwise, the previous status is unchanged. From an interactive shell, **exit** sets the *status* if specified, but does not terminate the shell. The shell executes **exit** directly.

See Also

commands, ksh, sh

exit() — General Function (libc)

Terminate a program gracefully
void exit(*status***) int** *status***;**

exit is the normal method to terminate a program directly. *status* information is passed to the parent process. By convention, an exit status of zero indicates success, whereas an exit status greater than zero indicates failure. If the parent process issued a **wait** call, it is notified of the termination and is passed the least significant eight bits of *status*. As **exit** never returns, it is always successful. Unlike the related function **_exit**, **exit** does extra cleanup, such as flushing buffered files and closing open files.

Example

For an example of this function, see the entry for **fopen**.

See Also

_exit(), close(), general functions, wait()

Notes

If a program leaves **main()** by an error condition, contents of register **AX** becomes the exit code. Usually, these register contents are random. If you want to test a program's return code, you must to exit or return from **main()**.

exp() — Mathematics Function (libm)

Compute exponent
#include <math.h>
double exp(*z***) double** *z***;**

exp returns the exponential of *z*, or *e*^*z*.

Example

The following program prompts you for a number, then prints the value for it as returned by **exp**, **pow**, **log**, and **log10**.

```
#include <math.h>
#include <stdio.h>
#define display(x) dodisplay((double)(x), #x)

dodisplay(value, name)
double value; char *name;
{
     if (errno)
          perror(name);
     else
          printf("%10g %s\n", value, name);
     errno = 0;
}

main()
{
     extern char *gets();
     double x;
     char string[64];

     for(;;) {
          printf("Enter number: ");
          if(gets(string) == NULL)
               break;
          x = atof(string);
```

```
            display(x);
            display(exp(x));
            display(pow(10.0,x));
            display(log(exp(x)));
            display(log10(pow(10.0,x)));
        }
}
```

See Also

errno, mathematics library

Diagnostics

exp indicates overflow by an **errno** of **ERANGE** and a huge returned value.

export — Command

Add a shell variable to the environment
export [*name* ...]
export [*name=value*]

When the shell executes a command, it passes the command an *environment*. By convention, the environment consists of assignments, each of the form *name=value*. For example, typing

```
export TERM=vt100
```

sets the environmental variable **TERM** to equal the string **vt100**.

A command may look for information in the environment or may simply ignore it. In the above example, a program that reads the variable **TERM** (such as MicroEMACS) will assume that you are working on a DEC VT-100 terminal or one that emulates it.

The shell places the *name* and the value of each shell variable that appears in an **export** command into the environment of subsequently executed commands. It does not place a shell variable into the environment until it appears in an **export** command.

With no arguments, **export** prints the name and the value of each shell variable currently marked for export.

The shell executes **export** directly.

See Also

commands, environ, exec, ksh, sh

expr — Command

Compute a command line expression
expr *argument* ...

The arguments to **expr** form an expression. **expr** evaluates the expression and writes the result on the standard output. Among other uses, **expr** lets the user perform arithmetic in shell command files.

Each *argument* is a separate token in the expression. An argument has a logical value 'false' if it is a null string or has numerical value zero, 'true' otherwise. Integer arguments consist of an optional sign followed by a string of decimal digits. The range of valid integers is that of signed long integers. No check is made for overflow or illegal arithmetic operations. Floating point numbers are not supported.

The following list gives each **expr** operator and its meaning. The list is in order of increasing operator precedence; operators of the same precedence are grouped together. All operators associate left to right except the unary operators '!', '-', and '**len**', which associate right to left. The spaces shown are significant - they separate the tokens of the expression.

{ *expr1*, *expr2*, *expr3* }
: Return *expr2* if *expr1* is logically true, and *expr3* otherwise. Alternatively, { *expr1* , *expr2* } is equivalent to { *expr1* , *expr2* , **0** }.

expr1 **|** *expr2*
: Return *expr1* if it is true, *expr2* otherwise.

expr1 **&** *expr2*
: Return *expr1* if both are true, zero otherwise.

expr1 relation expr2
: Where *relation* is one of <, <=, >, >=, ==, or !=, return one if the *relation* is true, zero otherwise. The comparison is numeric if both arguments can be interpreted as numbers, lexicographic otherwise. The lexicographic comparison is the same as **strcmp** (see **string**).

expr1 **+** *expr2*

expr1 **-** *expr2*
: Add or subtract the integer arguments. The expression is invalid if either *expr* is not a number.

expr1 ***** *expr2*

expr1 **/** *expr2*

expr1 **%** *expr2*
: Multiply, divide, or take remainder of the arguments. The expression is invalid if either *expr* is not numeric.

expr1 **:** *expr2*
: Match patterns (regular expressions). *expr2* specifies a pattern in the syntax used by **ed**. It is compared to *expr1*, which may be any string. If the \(...\) pattern occurs in the regular expression the matching operator returns the matched field from the string; if there is more than one \(...\) pattern the extracted fields are concatenated in the result. Otherwise, the matching operator returns the number of characters matched.

len *expr*
: Return the length of *expr*. It behaves like **strlen** (see **string**). *len* is a reserved word in **expr**.

!*expr* Perform logical negation: return zero if *expr* is true, one otherwise.

-expr Unary minus: return the negative of its integer argument. If the argument is non-numeric the expression is invalid.

(*expr*)
: Return the *expr*. The parentheses allow grouping expressions in any desired way.

The following operators have special meanings to the shell **sh**, and must be quoted to be interpreted correctly: **{ } ( ) < > & | ***.

See Also

commands, ed, sh, test

Notes

expr returns zero if the expression is true, one if false, and two if an error occurs. In the latter case an error message is also printed.

extern — C Keyword

Declare storage class

extern indicates that a C element belongs to the *external* storage class. Both variables and functions may be declared to be **extern**. Use of this keyword tells the C compiler that the variable or function is defined outside of the present file of source code. All functions and variables defined outside of functions are implicitly **extern** unless declared **static**.

When a source file references data that are defined in another file, it must declare the data to be **extern**, or the linker will return an error message of the form:

```
undefined symbol name
```

For example, the following declares the array **tzname**:

```
extern char tzname[2][32];
```

When a function calls a function that is defined in another source file or in a library, it should declare the function to be **extern**. In the absence of a declaration, **extern** functions are assumed to return **ints**, which may cause serious problems if the function actually returns a 32-bit pointer (such as on the 68000 or i8086 LARGE model), a **long**, or a **double**.

For example, the function **malloc** appears in a library and returns a pointer; therefore, it should be declared as follows:

```
extern char *malloc();
```

If you do not do so, the compiler assumes that **malloc** returns an **int**, and generate the error message

```
integer pointer pun
```

when you attempt to use **malloc** in your program.

See Also

auto, C keywords, pun, register, static, storage class

F

fabs() — Mathematics Function (libm)

Compute absolute value
#include <math.h>
double fabs(*z*) double *z*;

fabs implements the absolute value function. It returns *z* if *z* is zero or positive, or -*z* if *z* is negative.

Example

For an example of this function, see the entry for **ceil**.

See Also

abs(), ceil(), floor(), frexp(), mathematics library

factor — Command

Factor a number
factor [*number* ...]

factor computes and prints the prime factorials for each of a list of given *numbers*. If no *numbers* are given on the command line, **factor** reads numbers from the standard input.

See Also

commands

false — Command

Unconditional failure
false

false does nothing. It is guaranteed to fail. It can be useful in shell scripts, to force certain situations to occur.

See Also

commands, ksh, sh, true

Notes

Under the Korn shell, **false** is an alias for its built-in command **let**.

fblk.h — Header File

Define the disk-free block
#include <sys/fblk.h>

fblk.h defines the disk-free block **fblk**.

See Also

header files

fc — Command

Edit and re-execute one or more previous commands
fc [**-ln**] [*first* [*last*]]
fc -s [*old=new*] [*command*]"

fc, the "fix command", is a command built into the Korn shell **ksh**. It permits you to edit and re-execute one or more commands that have been executed previously.

fc has two forms, as shown above. The first version selects commands *first* through *last* and inserts them into a text editor. You can edit the commands in the editor; exiting from the editor re-executes the edited commands.

first and *last* can be addressed either by the command's number (the first command issued to the shell is number one, the second is number two, and so on), or by a string that matches the beginning of the command. The editor used is the one set in the environmental variable **FCEDIT** (default, **ed**).

When called without a *last* variable, the command selects just *first*. Option **-l** prints the commands on the standard output rather than buffering the commands for editing and re-execution. Option **-n** suppresses the default command numbers.

The second form of the **fc** command substitutes string *new* for string *old* within *command*, then re-executes it. *command* can be addressed either by its number or by a string that matches its beginning. If no *command* is specified, it re-executes the previous command.

See Also

commands, FCEDIT, ksh

FCEDIT — Environmental Variable

Editor used by fc command

The Korn shell's command **fc** reads the environmental variable **FCEDIT** to see which editor it should use to edit commands.

See Also

environmental variables, ksh

fclose() — STDIO Function (libc)

Close a stream
#include <stdio.h>
int fclose(*fp*) FILE **fp*;

fclose closes the stream **fp**. It calls **fflush** on the given *fp*, closes the associated file, and releases any allocated buffer. The function **exit** calls **fclose** for open streams.

Example

For examples of how to use this function, see the entries for **fopen** and **fseek**.

See Also

STDIO

Diagnostics

fclose returns **EOF** if an error occurs.

fcntl() — COHERENT System Call

Control open files
#include <sys/fcntl.h>
int fcntl(*fd*, *command*, *arg*)
int fd, cmd;

The COHERENT system call **fcntl** permits manipulation of an open file. *fd* is the file descriptor; this description must have been obtained from a call to **creat**, **dup**, **fcntl**, **open**, or **pipe**.

command identifies the task that you want **fcntl** to perform. *arg* is an argument specific to

the given *command*. **fcntl** recognizes the following commands:

F_DUPFD
Duplicate file descriptor *fd* onto the first available file descriptor greater than or equal to *arg*.

F_GETFL
Get the file flags for the file specified by *fd*.

F_SETFL
Set file flags for file descriptor *fd* to the value specified by *arg*.

fcntl returns the file descriptor **fd**.

See Also

COHERENT system calls, creat(), dup(), fcntl.h, file, file descriptor, open(), pipe()

fcntl.h — Header File

Manifest constants for file-handling functions
#include <sys/fcntl.h>

fcntl.h declares manifest constants that are used by the file-handling functions **open** and **fcntl**.

See Also

header files

fd — Device Driver

Floppy disk driver

The files **/dev/f*** are entries for the diskette drives of COHERENT on the IBM AT. Each entry is assigned major device number 4, is accessed as a block-special device, and has a corresponding character-special device entry.

The device entries are linked to a driver that handles up to four 5.25 inch disk drives, each in one of several formats. The least-significant four bits of an entry's minor device number identify the type of drive. The next least-significant two bits identify the drive. The following table summarizes the name, minor device number, sectors per track, partition sector size, characteristics, and addressing method for each device entry of floppy disk drive 0.

9 sectors/track					
f9d0	4	9	720	DSDD	surface (5.25 inch)
fqa0	13	9	1440	DSQD	cylinder (3.25 inch)
f9a0	12	9	720	DSDD	cylinder (5.25 inch)
15 sectors/track					
fha0	14	15	2400	DSHD	cylinder (5.25 inch)
18 sectors/track					
fva0	15	18	2880	DSHD	cylinder

Prefixing an **r** to a name given above gives the name of the corresponding character-device entry. Corresponding device entries for drives 1, 2, and 3 have minor numbers with offsets of 16, 32, and 48 from the minor numbers given above and have 1, 2, or 3 in place of 0 in the names given above.

For device entries whose minor number's fourth least-significant bit is zero (minor numbers 0 through 7 for drive 0), the driver uses surface addressing rather than cylinder addressing. This means that it increments tracks before heads when computing sector addresses and the first surface is used completely before the second surface is accessed. For devices whose minor number's fourth least significant bit is 1 (minor numbers 8 through 15 for drive 0), the driver uses cylinder addressing.

For a diskette to be accessible from the COHERENT system, a device file must be present in directory **/dev** with the appropriate type, major and minor device numbers, and permissions. The command **mknod** creates a special file for a device.

Files

<fdioctl.h> — Driver command header file
/dev/fd* — Block-special files
/dev/rfd* — Character special files

See Also

device drivers, fdformat, mkfs, mknod,

Diagnostics

The driver reports any error status received from the controller and retries the operation several times before it reports an error to the program that initiated an operation.

Notes

The driver assumes that the disk is formatted with eight, nine, 15, or 18 sectors of 512 bytes each per track, depending upon the **/dev** entry. Cylinder addressing is the norm for COHERENT.

Programs that use the raw device interface must read whole sectors into buffers that do not straddle DMA boundaries.

fd.h — Header File

Declare file-descriptor structure
#include <sys/fd.h>

fd.h declares the file-descriptor structure **fd**, plus associated constants and the function **fdget**.

See Also

header files

fdformat — Command

Format a floppy disk
/etc/fdformat [*option* ...] *special*

fdformat formats a floppy disk. The given *special* should be the name of the special file that correspond to the floppy disk drive.

fdformat recognizes the following options:

-a Print information on the standard output device during format. As it formats a cylinder, it will print a line of the form

```
hd=0 cyl=25
```

on your screen.

-i *number*
Use *number* (0 through 7) as the interleave factor in formatting. Note that the default interleave is six.

-o *number*
Use *number* (default, 0) as the skew factor for sector numbering.

-v Verify formatting and verify data written with the **-w** option.

-w *file* Format the floppy disk and then copy *file* to it track by track. The raw device should be used.

The command **mkfs** builds a COHERENT file system on a formatted floppy disk. The command **dos** builds a DOS file system on a formatted floppy disk and transfers files to or from it. The command **mount** mounts a floppy disk containing a file system to allow access to it through the COHERENT directory structure. The command **umount** unmounts a floppy disk.

Examples

The following command formats a 2400-block, 5.25-inch floppy disk in drive 0 (otherwise known known as drive A):

```
/etc/fdformat -v /dev/fha0
```

The following command formats a 1440-block, 3.5-inch floppy disk in drive 1 (otherwise known as drive B):

```
/etc/fdformat -v /dev/fqa1
```

See Also

commands, dos, fd, mkfs, mount, umount

Diagnostics

When errors occur on floppy-disk devices, the driver prints on the system console an error message that describes the error.

Notes

fdformat formats a track at a time. **fdformat** can be interrupted between tracks, which may result in a partially formatted floppy disk.

fdioctl.h — Header File

Control floppy-disk I/O
#include <sys/fdioctl.h>

fdioctl.h declares constants and structures used to control floppy-disk I/O.

See Also

header files

fdisk — Command

Hard-disk partitioning utility
/etc/fdisk [-r] [-c] [-b *mboot*] *xdev* ...

The COHERENT version of the command **fdisk** supports flexible hard-disk partitioning among four operating systems, i.e. MS-DOS, CP/M, COHERENT, and XENIX. This capability means that with the COHERENT **fdisk**, you can support COHERENT plus any combination of MS-DOS, CP/M, or XENIX on one hard disk.

fdisk recognizes the following flags:

-r Read-only access to partitioning information.

-b Use the first 446 bytes of *mboot* as master boot code to replace that in *xdev*.

-c Allow the specification of disk geometry (i.e., number of cylinders, heads, sectors) for disk drives that are not supported by the system BIOS.

fdisk accesses the first block from the special device *xdev* (e.g., /dev/at0x) for the partitioning information. **fdisk** then queries the user for changes. These changes are written to *xdev* only if the user requests the changes to be saved. If omitted, *xdev* defaults to **/dev/at0x** and **/dev/at1x**. SCSI disk device users will need to specify *xdev* as **/dev/sd***n***x** where *n* is a digit corresponding to the SCSI ID for the disk device (e.g., **/dev/sd0x**).

Files

<fdisk.h>

See Also

commands

Notes

If the partition table is changed, the system should be rebooted; most device drivers will not recognize the revised partition information until a reboot occurs.

As the **-r** and **-b** options are contradictory, attempting to use them together generates an error message.

Please note that some versions of **fdisk** for other operating systems can rearrange the order of entries in the partition table. If this happens, you may lose the ability to run COHERENT until the table is restored to its previous order. A sign of this problem is getting the prompt **AT boot?** when trying to start COHERENT after running any **fdisk** program, and not being able to get past it.

Computer systems that use older BIOS releases may report incorrect disk parameters. Users of such systems should change the CMOS setup values if possible, but the BIOS on some older systems will not allow you to specify arbitrary values for disk parameters. Users with such systems can use the **fdisk -c** option instead.

If you plan to install and run COHERENT and MS-DOS on the same hard disk, note the following:

- If you wish to install COHERENT and MS-DOS on the same hard drive, you *must* run the MS-DOS **fdisk** first!
- If you plan on running both operating systems, you *must* install MS-DOS first and leave some free cylinders on the disk for COHERENT as well as a free partition. You can have both primary as well as extended MS-DOS partitions on the same drive as COHERENT, but COHERENT cannot use a sub-partition of the MS-DOS extended partition. COHERENT must have one of the four *real* partitions.

Failure to observe these rules will result in loss of data

fdisk.h — Header File

Fixed-disk constants and structures
#include <sys/fdisk.h>

fdisk.h declares structures and constants used to manipulate the fixed disk.

See Also

header files

fdopen() — STDIO Function (libc)

Open a stream for standard I/O
#include <stdio.h>
FILE *fdopen(*fd, type***) int** *fd*; **char** **type*;

fdopen allocates and returns a **FILE** structure, or *stream*, for the file descriptor *fd*, as obtained from **open**, **creat**, **dup**, or **pipe**. *type* is the manner in which you want *fd* to be opened, as follows:

r	Read a file
w	Write into a file
a	Append onto a file

Example

The following example obtains a file descriptor with **open**, and then uses **fdopen** to build a pointer to the **FILE** structure.

```
#include <ctype.h>
#include <stdio.h>

void adios(message)
char *message;
{
     fprintf(stderr, "%s\n", message);
     exit(1);
}

main(argc, argv)
int argc; char *argv[];
{
     extern FILE *fdopen();
     FILE *fp;
     int fd;
     int holder;

     if (--argc != 1)
          adios("Usage: example filename");

     if ((fd = open(argv[1], 0)) == -1)
          adios("open failed.");
     if ((fp = fdopen(fd, "r")) == NULL)
          adios("fdopen failed.");
```

```
        while ((holder = fgetc(fp)) != EOF) {
                if ((holder > '\177') && (holder < ' '))
                        switch(holder) {
                        case '\t':
                        case '\n':
                                break;
                        default:
                                fprintf(stderr, "Seeing char %d\n", holder);
                                exit(1);
                        }
                fputc(holder, stdout);
        }
}
```

See Also

creat(), dup(), fopen(), open(), STDIO

Diagnostics

fdopen returns NULL if it cannot allocate a **FILE** structure. Currently, only 20 **FILE** structures can be allocated per program, including **stdin**, **stdout**, and **stderr**.

feof() — STDIO Macro (stdio.h)

Discover stream status
#include <stdio.h>
int feof(*fp*) FILE **fp*;

feof is a macro that tests the status of the argument stream *fp*. It returns a number other than zero if *fp* has reached the end of file, and zero if it has not. One use of **feof** is to distinguish a value of -1 returned by **getw** from an **EOF**.

Example

For an example of how to use this function, see the entry for **fopen**.

See Also

EOF, STDIO

ferror() — STDIO Macro (stdio.h)

Discover stream status
#include <stdio.h>
int ferror(*fp*) FILE **fp*;

ferror is a macro that tests the status of the file stream *fp*. It returns a number other than zero if an error has occurred on *fp*. Any error condition that is discovered will persist either until the stream is closed or until **clearerr** is used to clear it. For write routines that employ buffers, **fflush** should be called before **ferror**, in case an error occurs on the last block written.

Example

This example reads a word from one file and writes it into another.

```
#include <stdio.h>
```

```
main()
{
      FILE *fpin, *fpout;
      int inerr = 0;
      int outerr = 0;
      int word;
      char infile[20], outfile[20];

      printf("Name data file you wish to copy:\n");
      gets(infile);
      printf("Name new file:\n");
      gets(outfile);

      if ((fpin = fopen(infile, "r")) != NULL) {
            if ((fpout = fopen(outfile, "w")) != NULL) {

                  for (;;) {
                        word = fgetw(fpin);
                        if (ferror(fpin)) {
                              clearerr(fpin);
                              inerr++;
                        }

                        if (feof(fpin))
                              break;
                        fputw(word, fpout);
                        if (ferror(fpout)) {
                              clearerr(fpout);
                              outerr++;
                        }
                  }

            } else {
                  printf
                        ("Cannot open output file %s\n",
                              outfile);
                  exit(1);
            }

      } else {
            printf("Cannot open input file %s\n", infile);
            exit(1);
      }

      printf("%d - read error(s)  %d - write error(s)\n",
            inerr, outerr);
      exit(0);
}
```

See Also

STDIO

fflush() — STDIO Function (libc)

Flush output stream's buffer

#include <stdio.h>
int fflush(*fp*) FILE **fp*;

fflush flushes any buffered output data associated with the file stream *fp*. The file stream stays open after **fflush** is called. **fclose** calls **fflush**, so there is no need for you to call it when normally closing a file or buffer.

Example

This example demonstrates **fflush**. When run, you will see the following:

```
Line 1
-----
Line 1
-----
Line 1
Line 2
-----
```

The call

```
fprintf(fp, "Line 2\n");
```

goes to a buffer and is not in the file when file **foo** is listed. However if you redirect the output of this program to a file and list the file, you will see:

```
Line 1
Line 1
Line 1
Line 2
-----
-----
-----
```

because the line

```
printf("-----\n");
```

goes into a buffer and is not printed until the program is over and all buffers are flushed by **exit()**.

Although the COHERENT screen drivers print all output immediately, not all operating systems work this way, so when in doubt, **fflush()**.

```
#include <stdio.h>

main()
{
	FILE *fp;
```

```
        if (NULL == (fp = fopen("foo", "w")))
             exit(1);
        fprintf (fp, "Line 1\n");
        fflush (fp);
        system ("cat foo"); /* print Line 1 */

        printf("-----\n");
        fprintf(fp, "Line 2\n");
        system("cat foo"); /* print Line 1 */
        printf("-----\n");

        fflush(fp);
        system("cat foo"); /* print Line 1 Line 2 */
        printf("-----\n");
}
```

See Also

fclose(), setbuf(), STDIO, write()

Diagnostics

fflush returns **EOF** if it cannot flush the contents of the buffers; otherwise it returns a meaningless value.

Note, also, that all STDIO routines are buffered. **fflush** should be used to flush the output buffer if you follow a STDIO routine with an unbuffered routine.

fgetc() — STDIO Function (libc)

Read character from stream

#include <stdio.h>

int fgetc(*fp*) FILE **fp*;

fgetc reads characters from the input stream *fp*. In general, it behaves the same as the macro **getc**: it runs more slowly than **getc**, but yields a smaller object module when compiled.

Example

This example counts the number of lines and "sentences" in a file.

```
#include <stdio.h>

main()
{
     FILE *fp;
     int filename[20];
     int ch;
     int nlines = 0;
     int nsents = 0;

     printf("Enter file to test: ");
     gets(filename);
```

```
        if ((fp = fopen(filename,"r")) == NULL) {
             printf("Cannot open file %s.\n", filename);
             exit(1);
        }

        while ((ch = fgetc(fp)) != EOF) {
             if (ch == '\n')
                  ++nlines;

             else if (ch == '.' || ch == '!' || ch == '?') {
                  if ((ch = fgetc(fp)) != '.')
                        ++nsents;

                  else
                       while((ch=fgetc(fp)) == '.')
                             ;
                  ungetc(ch, fp);
             }
        }

        printf("%d line(s), %d sentence(s).\n",
             nlines, nsents);
}
```

See Also

getc(), STDIO

Diagnostics

fgetc returns EOF at end of file or on error.

fgets() — STDIO Function (libc)

Read line from stream
#include <stdio.h>
char *fgets(*s*, *n*, *fp***) char ***s*; **int** *n*; **FILE ***fp*;

fgets reads characters from the stream *fp* into string *s* until either *n*-1 characters have been read, or a newline or EOF is encountered. It retains the newline, if any, and appends a null character at the end of of the string. **fgets** returns the argument *s* if any characters were read, and NULL if none were read.

Example

This example looks for the pattern given by **argv[1]** in standard input or in file **argv[2]**. It demonstrates the functions **pnmatch**, **fgets**, and **freopen**.

```
#include <stdio.h>
#define MAXLINE 128
char buf[MAXLINE];

void fatal(s) char *s;
{
     fprintf(stderr, "pnmatch: %s\n", s);
     exit(1);
}
```

```
main(argc, argv)
int argc; char *argv[];
{
     if (argc != 2 && argc != 3)
          fatal("Usage: pnmatch pattern [ file ]");

     if (argc==3 && freopen(argv[2], "r", stdin)==NULL)
          fatal("cannot open input file");

     while (fgets(buf, MAXLINE, stdin) != NULL) {
          if (pnmatch(buf, argv[1], 1))
               printf("%s", buf);
     }

     if (!feof(stdin))
          fatal("read error");
     exit(0);
}
```

See Also

fgetc(), gets(), STDIO

Diagnostics

fgets returns NULL if an error occurs, or if **EOF** is seen before any characters are read.

fgetw() — STDIO Function (libc)

Read integer from stream

#include <stdio.h>
int fgetw(*fp***) FILE ****fp***;

fgetw reads an integer from the stream *fp*.

Example

For an example of this function, see the entry for **ferror**.

See Also

fputw(), STDIO

Notes

fgetw returns EOF on errors. A call to **feof** or **ferror** may be necessary to distinguish this value from a genuine end-of-file signal.

field — Definition

A **field** is an area that is set apart from whatever surrounds it, and that is defined as containing a particular type of data. In the context of C programming, a field is either an element of a structure, or a set of adjacent bits within an **int**.

See Also

bit map, data formats, definitions, structure

file — Command

Guess a file's type
file *file* ...

file examines each *file* and takes an educated guess as to its type. **file** recognizes the following classes of text files: files of commands to the shell; files containing the source for a C program; files containing **yacc** or **lex** source; files containing assembly language source; files containing unformatted documents that can be passed to **nroff**; and plain text files that fit into none of the above categories.

file recognizes the following classes of non-text or binary data files: the various forms of archives, object files, and link modules for various machines, and miscellaneous binary data files.

See Also

commands, ls, size

Notes

Because **file** only reads a set amount of data to determine the class of a text file, mistakes can happen.

file — Definition

A **file** is a mass of bits that has been given a name and is stored on a nonvolatile medium. These bits may form ASCII characters or machine-executable data. Under the COHERENT system and related operating systems, external devices can mimic files, in that they can be opened, closed, read, and written to in a manner identical to that of files.

To manipulate the contents of a file, you must first open it. This can be done with the COHERENT system call **open**, or with the function **fopen**. You can then read the file, write material to it, or append material onto it with the COHERENT system calls **read** and write, or with the functions **fread** and **fwrite**. See the entries on **COHERENT system calls** and entry **STDIO** for more information on manipulating material within a file.

See Also

close(), definitions, executable file, fopen(), fclose(), FILE, open()

FILE — Definition

Descriptor for a file stream
#include <stdio.h>

FILE describes a *file stream* which can be either a file on disk or a peripheral device through which data flow. It is defined in the header file **stdio.h**.

A pointer to **FILE** is returned by **fopen**, **freopen**, **fdopen**, and related functions.

The **FILE** structure is as follows:

```
typedef struct  FILE
{
     unsigned char *_cp,
                   *_dp,
                   *_bp;
     int  _cc;
     int  (*_gt)(),
          (*_pt)();
     int  _ff;
     char _fd;
     int  _uc;
} FILE;
```

_cp points to the current character in the file. **_dp** points to the start of the data within the buffer. **_bp** points to the file buffer. **_cc** is the number of unprocessed characters in the buffer. **_gt** and **_pt** point, respectively, to the functions **getc** and **putc**. **_ff** is a bit map that holds the various file flags, as follows:

_FINUSE	0x01	Unused
_FSTBUF	0x02	Used by macro **setbuf**
_FUNGOT	0x04	Used by **ungetc**
_FEOF	0x08	Tested by macro **feof**
_FERR	0x10	Tested by macro **ferror**

_fd is the file descriptor, which is used by low-level routines like **open**; it is also used by **reopen**. Finally, **_uc** is the character that has been "ungotten" by **ungetc**, should it be used.

See Also

definitions, fopen(), freopen(), stdio.h, stream

file descriptor — Definition

A **file descriptor** is an integer between 1 and 20 that indexes an area in the operating system's list of internal file descriptors. It is used by routines like **open**, **close**, and **lseek** to work with files. A file descriptor is *not* the same as a **FILE** stream, which is used by routines like **fopen**, **fclose**, or **fread**.

See Also

COHERENT system calls, definitions, file, FILE

file formats — Overview

The COHERENT system uses a number of different file formats. Each format is designed to order most efficiently the information that that file holds. This manual describes the following special file formats:

core	Core dump file format
group	Group file format
L-dev	Describe devices used by UUCP
L.sys	Describe remote sites to UUCP
passwd	Password file format
Permissions	Format of UUCP permissions file
ttys	Active terminal ports

The following header files also hold information on file formats:

acct.h	Format for process-accounting file
ar.h	Format for archive files
canon.h	Portable layout of binary data
dir.h	Directory format
l.out.h	Object file format
mtab.h	Currently mounted file systems
utmp.h	Login accounting information

See their respective entries in this Lexicon for a fuller description of their contents.

See Also

header files, Lexicon

fileno() — STDIO Function (libc)

Get file descriptor

#include <stdio.h>
int fileno(*fp*) FILE **fp*;

fileno returns the file descriptor associated with the file stream *fp*. The file descriptor is the integer returned by **open** or **creat**. It is used by routines such as **fopen** to create a **FILE** stream.

Example

This example reads a file descriptor and prints it on the screen.

```
#include <stdio.h>

main(argc,argv)
int argc; char *argv[];
{
     FILE *fp;
     int fd;

     if (argc !=2) {
          printf("Usage: fd_from_fp filename\n");
          exit(0);
     }

     if ((fp = fopen(argv[1], "r")) == NULL) {
          printf("Cannot open input file\n");
          exit(0);
     }

     fd = fileno(fp);
     printf("The file descriptor for %s is %d\n",
          argv[1], fd);
}
```

See Also

FILE, file descriptor, STDIO

filsys.h — Header File

Structures and constants for super block
#include <sys/filsys.h>

filsys.h declares structures and constants used to by functions that manipulate the super block.

See Also

header files

filter — Definition

A *filter* is a program that reads a stream of input, transforms it in a precisely defined manner, and writes it to another stream. Two or more filters can be coupled with *pipes* to perform a complex transformation on a stream of input.

See Also

definitions, pipe

find — Command

Search for files satisfying a pattern
find *directory* ... [*expression* ...]

no -depth

find traverses each given *directory*, testing each file or subdirectory found with the *expression* part of the command line. The test can be the basis for deciding whether to process the file with a given command.

If the command line specifies no *expression* or specifies no execution or printing (**-print**, **-exec**, or **-ok**), by default **find** prints the pathnames of the files found.

In the following, *file* means any file: directory, special file, ordinary file, and so on. Numbers represented by *n* may be optionally prefixed by a '+' or '-' sign to signify values greater than *n* or less than *n*, respectively.

find recognizes the following *expression* primitives:

-atime *n* Match if the file was accessed in the last *n* days.

-ctime *n* Match if the i-node associated with the file was changed in the last *n* days, as by **chmod**.

-exec *command*
Match if *command* executes successfully (has a zero exit status). The *command* consists of the following arguments to **find**, terminated by a semicolon ';' (escaped to get past the shell). **find** substitutes the current pathname being tested for any argument of the form '{}'.

-group *name*
Match if the file is owned by group *name*. If *name* is a number, the owner must have that group number.

-inum *n* Match if the file is associated with i-number *n*.

-links *n* Match if the number of links to the file is *n*.

-mtime *n* Match if the most recent modification to the file was *n* days ago.

-name *pattern*
Match if the file name corresponds to *pattern*, which may include the special characters '*', '?', and '[...]' recognized by the shell **sh**. The *pattern* matches only

the part of the file name after any slash ('/') characters.

-newer *file* Match if the file is newer than *file*.

-nop Always match; does nothing.

-ok *command*
Same as **-exec** above, except prompt interactively and only executes *command* if the user types response **'y'**.

-perm *octal* Match if owner, group, and other permissions of the file are the *octal* bit pattern, as described in **chmod**. When *octal* begins with a **'-'** character, more of the permission bits (setuid, setgid, and sticky bit) become significant.

-print Always match; print the file name.

-size *n* Match if the file is *n* blocks in length; a block is 512 bytes long.

-type *c* Match if the type of the file is *c*, chosen from the set **bcdfmp** (for block special, character special, directory, ordinary file, multiplexed file, or pipe, respectively).

-user *name* Match if the file is owned by user *name*. If *name* is a number, the owner must have that user number.

exp1 exp2 Match if both expressions match. **find** evaluates *exp2* only if *exp1* matches.

exp1 **-a** *exp2*
Match if both expressions match, as above.

exp1 **-o** *exp2*
Match if either expression matches. **find** evaluates **exp2** only if **exp1** does not match.

! *exp* Match if the expression does *not* match.

(*exp*) Parentheses are available for expression grouping.

Examples

A **find** command to print the names of all files and directories in user **fred**'s directory is:

```
find /usr/fred
```

The following, more complicated **find** command prints out information on all **core** and object (**.o**) files that have not been changed for a day. Because some characters are special both to **find** and **sh**, they must be escaped with '\' to avoid interpretation by the shell.

```
find / \( -name core -o -name \*.o \) -mtime +1 \
-exec ls -l {} \;
```

See Also

chmod, commands, ls, sh, srcpath, test

fixstack — Command

Change stack allocation
fixstack +-*value* [*filename*]

fixstack alters the stack size of an executable file. It enlarges or shrinks the stack by *value* bytes. *value* is assumed to be a hexadecimal number, and must be preceded by + or -.

If *filename* is not given, **fixstack** by default alters the stack size of file **l.out**.

See Also

cc, commands, size

float — C Keyword

Data type

Floating point numbers are a subset of the real numbers. Each has a built-in radix point (or "decimal point") that shifts, or "floats", as the value of the number changes. It consists of the following: one sign bit, which indicates whether the number is positive or negative; bits that encode the number's *exponent*; and bits that encode the number's *fraction*, or the number upon which the exponent works. In general, the magnitude of the number encoded depends upon the number of bits in the exponent, whereas its precision depends upon the number of bits in the fraction.

The exponent often uses a *bias*. This is a value that is subtracted from the exponent to yield the power of two by which the fraction will be increased.

Floating point numbers come in two levels of precision: single precision, called **floats**; and double precision, called **doubles**. With most microprocessors, **sizeof(float)** returns four, which indicates that it is four **chars** (bytes) long, and **sizeof(double)** returns eight.

Several formats are used to encode **floats**, including IEEE, DECVAX, and BCD (binary coded decimal). COHERENT uses DECVAX format throughout.

The following describes DECVAX, IEEE, and BCD formats, for your information.

DECVAX Format

The 32 bits in a **float** consist of one sign bit, an eight-bit exponent, and a 24-bit fraction, as follows. Note that in this diagram, 's' indicates "sign", 'e' indicates "exponent", and 'f' indicates "fraction":

```
=============
| seee eeee | Byte 4
|===========|
| efff ffff | Byte 3
|===========|
| ffff ffff | Byte 2
|===========|
| ffff ffff | Byte 1
=============
```

The exponent has a bias of 129.

If the sign bit is set to one, the number is negative; if it is set to zero, then the number is positive. If the number is all zeroes, then it equals zero; an exponent and fraction of zero plus a sign of one ("negative zero") is by definition not a number. All other forms are numeric values.

The most significant bit in the fraction is always set to one and is not stored. It is usually called the "hidden bit".

The format for **doubles** simply adds another 32 fraction bits to the end of the **float** representation, as follows:

```
============
| seee eeee |Byte 8
|===========|
| efff ffff |Byte 7
|===========|
| ffff ffff |Byte 6
|==========|
| ffff ffff |Byte 5
|===========|
| ffff ffff |Byte 4
|===========|
| ffff ffff |Byte 3
|===========|
| ffff ffff |Byte 2
|===========|
| ffff ffff |Byte 1
============
```

IEEE Format

The IEEE encoding of a **float** is the same as that in the DECVAX format. Note, however, that the exponent has a bias of 127, rather than 129.

Unlike the DECVAX format, IEEE format assigns special values to several floating point numbers. Note that in the following description, a *tiny* exponent is one that is all zeroes, and a *huge* exponent is one that is all ones:

- A tiny exponent with a fraction of zero equals zero, regardless of the setting of the sign bit.
- A huge exponent with a fraction of zero equals infinity, regardless of the setting of the sign bit.
- A tiny exponent with a fraction greater than zero is a denormalized number, i.e., a number that is less than the least normalized number.
- A huge exponent with a fraction greater than zero is, by definition, not a number. These values can be used to handle special conditions.

An IEEE **double**, unlike DECVAX format, increases the number of exponent bits. It consists of a sign bit, an 11-bit exponent, and a 53-bit fraction, as follows:

```
=============
| seee eeee |    Byte 8
|===========|
| eeee ffff |    Byte 7
|===========|
| ffff ffff |    Byte 6
|===========|
| ffff ffff |    Byte 5
|===========|
| ffff ffff |    Byte 4
|===========|
| ffff ffff |    Byte 3
|===========|
| ffff ffff |    Byte 2
|===========|
| ffff ffff |    Byte 1
=============
```

The exponent has a bias of 1,023. The rules of encoding are the same as for **floats**.

BCD Format

The BCD format ("binary coded decimal", also called "packed decimal") is used to eliminate rounding errors that alter the worth of an account by a fraction of a cent. It consists of a sign, an exponent, and a chain of four-bit numbers, each of which is defined to hold the values zero through nine.

A BCD **float** has a sign bit, seven bits of exponent, and six four-bit digits. In the following diagrams, 'd' indicates "digit":

```
=============
| seee eeee |    Byte 4
|===========|
| dddd dddd |    Byte 3
|===========|
| dddd dddd |    Byte 2
|===========|
| dddd dddd |    Byte 1
=============
```

A BCD **double** has a sign bit, 11 bits of exponent, and 13 four-bit digits, as follows:

```
=============
| seee eeee |    Byte 8
|===========|
| eeee dddd |    Byte 7
|===========|
| dddd dddd |    Byte 6
|===========|
| dddd dddd |    Byte 5
|===========|
| dddd dddd |    Byte 4
|===========|
| dddd dddd |    Byte 3
|===========|
| dddd dddd |    Byte 2
|===========|
| dddd dddd |    Byte 1
=============
```

Passing the hexadecimal numbers A through F in a digit yields unpredictable results.

The following rules apply when handling BCD numbers:

- A tiny exponent with a fraction of zero equals zero.
- A tiny exponent with a fraction of non-zero indicates a denormalized number.
- A huge exponent with a fraction of zero indicates infinity.
- A huge exponent with a fraction of non-zero is, by definition, not a number; these non-numbers are used to indicate errors.

See Also

C keywords, data formats, double
The Art of Computer Programming, vol. 2, page 180*ff*

floor() — Mathematics Function (libm)

Set a numeric floor
#include <math.h>
double floor(*z***) double** *z***;**

floor sets a numeric floor. It returns a double-precision floating point number whose value is the largest integer less than or equal to *z*.

Example

For an example of this function, see the entry for **ceil**.

See Also

abs(), ceil(), fabs(), frexp(), mathematics library

floppy disks — Technical Information

The COHERENT system lets you read or write to floppy disks, using a variety of different formats. You can choose the format that best suits the task at hand.

Disks Supported

COHERENT lets you use either 3.5-inch or 5.25-inch disks, in either high or low density; what you use depends upon the type of hardware that you have. The following table gives some commonly used diskette device names and formats:

Device name	*Sectors/Track*	*Heads*	*Sectors*	*Bytes*	*Format*
/dev/f9a0	9	2	720	360 KB	5.25"
/dev/f9a1	9	2	720	360 KB	5.25"
/dev/fqa0	9	2	1440	720 KB	3.5"
/dev/fqa1	9	2	1440	720 KB	3.5"
/dev/fha0	15	2	2400	1.2 MB	5.25"
/dev/fha1	15	2	2400	1.2 MB	5.25"
/dev/fva0	18	2	2880	1.44 MB	3.5"
/dev/fva1	18	2	2880	1.44 MB	3.5"

Device names ending in '0' indicate drive A:, names ending in '1' indicate drive B:. For a fuller description of COHERENT's floppy-disk devices, see the Lexicon entry for **fd**.

MS-DOS Format

COHERENT lets you read or write to floppy disks that contain MS-DOS file systems. Both tasks use the command **dos**. This command is discussed in full in its Lexicon entry.

To read files from a MS-DOS disk, use the **dos** command's **-x** option, with the appropriate device name for the floppy-disk device that you will be using (as given in the above table). For example, to read file **fred.exe** from a low-density, 5.25-inch MS-DOS floppy disk in drive A, use the following command:

```
dos -x /dev/f9a0 fred.exe
```

The following command reads all files from a high-density, 5.25-inch MS-DOS floppy disk in drive B:

```
dos -x /dev/fha1
```

To write a file to a preformatted MS-DOS floppy disk, use the **-r** option to the MS-DOS command. For example, to write file **fred.ms**, which contains text, to a low-density, 5.25-inch MS-DOS floppy disk in drive A, use the following command:

```
dos -ra /dev/f9a0 fred.ms
```

Note that the 'a' flag in the command line tells COHERENT to convert linefeeds to the linefeed/carriage return combination, as used by MS-DOS. You will want to use this flag *only* when transferring text files to or from an MS-DOS floppy disk.

The following command copies all files in the current directory to a high-density, 3.5-inch MS-DOS floppy disk in drive B:

```
dos -r /dev/fva1
```

Note that when you copy a file to an MS-DOS floppy disk, COHERENT observes the MS-DOS file-name conventions: it permits only eight characters to the left of the period, and only three characters to the right of it.

(It should be noted in passing that you can use the **dos** command to read files from or write files to an MS-DOS partition on your hard disk. All that is necessary is to replace the name of floppy-disk device with that of the hard-disk device for the partition in question. See the Lexicon entry for **at** for a list of hard-disk devices; see the entry for **fdisk** for information on how to read the layout of your hard disk; and see the entry for **dos** for details of how to use

the command.)

Finally, COHERENT lets you format a floppy disk and create an MS-DOS file system on it. To do so, you must use the command **fdformat** as well as **dos**. **fdformat** is described in detail in its Lexicon article.

To format a high-density, 5.25-inch floppy disk in drive B and write an MS-DOS file system onto it, use the following commands:

```
/etc/fdformat -av /dev/fha1
dos -F /dev/fha1
```

To write files to your newly created file system, use the **-r** option to the **dos** command, as described above.

COHERENT Format

If you wish, you can create a COHERENT file system on a floppy disk, mount it, and manipulate the files on it with standard COHERENT commands. This is a good illustration of the fact that to COHERENT a file system is a file system, whether it resides on a hard, a floppy disk, or any other mass-storage device. You can use such mountable floppy disks as an easy method of backing up files, or as a flexible extension to any other file system that you have currently mounted.

To create a COHERENT file system on a floppy disk, you must use the commands **fdformat**, **badscan**, and **mkfs**. Each is described in detail in its own Lexicon article. The following example creates a COHERENT file system on a high-density, 3.5-inch floppy disk placed in drive B:

```
/etc/fdformat -a /dev/fva1
/etc/badscan -v -o proto /dev/fva1 2880
/etc/mkfs /dev/fva1 proto
rm proto
```

In this example, command **fdformat** formatted the disk. **badscan** then scanned the disk for any bad blocks, and wrote its results into file **proto**. Finally, command **mkfs** reads **proto** and used its contents to create a COHERENT file system on the disk.

Now that the file system is created on the disk, you must mount it. While it is customary to mount file systems under directory '/', you are not required to do it. For example, if your login identifier is **fred** and your home directory is **/usr/fred**, you can mount the floppy disk's file system onto a subdirectory of **/usr/fred** and so make the floppy disk, in effect, an extension of your home directory. The following command does this for the 3.5-inch disk we formatted in the above example:

```
/etc/mount /dev/fva1 /usr/fred/temp
```

Now, all files you copy into directory **/usr/fred/temp** using the **cp** command will be written directly onto the floppy disk. Note that you may need to log in as the superuser **root** and use the command **chown** to ensure that **fred** owns the file system on that floppy disk. For details on **chown**, see its entry in the Lexicon. For details on shorthand notations for **mount**, see its entry in the Lexicon.

One last point about mounting file systems: before you remove a COHERENT-formatted floppy disk from its drive, you **must** first use the command **/etc/umount** to unmount its file system. If you do not, all data that COHERENT has stored in its buffers will not be written to the disk, and may be lost. Worse, if you remove one COHERENT disk and insert another without unmounting the old disk and mounting the new one, COHERENT will write all data in its buffers onto the new disk without regard for what that disk contains; in all

likelihood, this will trash the file system on the new disk and render its data unreadable. So, the lesson is: *always unmount a floppy disk before you remove it!*. To unmount the floppy disk we used in our previous example, use the command:

```
/etc/umount /dev/fva1
```

By the way, that's not a misprint: the command is **umount**, not "unmount".

Raw Format

Finally, COHERENT lets you use floppy disks in their raw form as a backup medium, much as you would use magnetic tape on a larger computer. You must first use the command **fdformat** with the **-v** option to format the floppy disks you will be using; it is also wise to label and number the disks so you can keep them in some reasonable order. Then you can use any of COHERENT's archiving utilities, such as **ustar**, **cpio**, or **dump** to archive directories or entire file systems onto the disks. It is recommended that you format a generous supply of floppy disks before you begin; if you run short of disks while archiving your files, you will have to abort, format more disks, and begin again. For details on how to use the archiving programs, see their respective entries in the Lexicon.

See Also

badscan, cpio, dos, dump, fd, fdformat, mkfs, mount, technical information, umount, ustar

fnkey — Command

Set/print function keys for the console
fnkey [*n* [*string*]]

The console keyboard of an AT COHERENT system includes ten programmable function keys, labeled **F1** through **F10**. Initially, these are programmed to send the escape sequences set by the **nkb** keyboard driver.

fnkey with a numeric argument programs function key **F***n* to send the given *string*, where *n* is a number from one through ten. If no *string* is given, **fnkey** resets **F***n* to send nothing.

With no argument, **fnkey** prints the current string for each programmed function key.

fnkey also lets you change the default bindings for other special or function keys. See Lexicon articles **keyboard tables** and **nkb** for details.

Example

To set function key **F2** to execute the COHERENT command **date**, use the following command:

```
fnkey 2 'date
'
```

Note that this command sets **F2** to the string **date\n**. If you type **fnkey** without any arguments, it displays the binding of all function keys including the following:

```
F2:  date\n
```

Files

/dev/console

See Also

commands, keyboard tables, nkb

Diagnostics

fnkey prints "cannot open /dev/console" if you lack permission to open **/dev/console**.

fopen() — STDIO Function (libc)

Open a stream for standard I/O
#include <stdio.h>
FILE *fopen (*name*, *type*) **char** **name*, **type*;

fopen allocates and initializes a **FILE** structure, or *stream*; opens or creates the file *name*; and returns a pointer to the structure for use by other **STDIO** routines. *name* refers to the file to be opened. *type* is a string that consists of one or more of the characters "rwa", to indicate the mode of the string, as follows:

r	Read; error if file not found
w	Write; truncate if found, create if not found
a	Append to end of file; no truncation, create if not found
r+	Read and write; no truncation, error if not found
w+	Write and read; truncate if found, create if not found
a+	Append and read; no truncation, create if not found

The modes that contain 'a' set the seek pointer to point at the end of the file; all other modes set it to point at the beginning of the file. Modes that contain '+' both read and write; however, a program must call **fseek** or **rewind** before it switches from reading to writing or vice versa.

Example

This example copies **argv[1]** to **argv[2]** using STDIO routines. It demonstrates the functions **fopen**, **fread**, **fwrite**, **fclose**, and **feof**.

```
#include <stdio.h>
/* BUFSIZ is defined in stdio.h */
char buf[BUFSIZ];

void fatal(message)
char *message;
{
     fprintf(stderr, "copy: %s\n", message);
     exit(1);
}

main(argc, argv)
int argc; char *argv[];
{
     register FILE *ifp, *ofp;
     register unsigned int n;

     if (argc != 3)
          fatal("Usage: copy source destination");
     if ((ifp = fopen(argv[1], "r")) == NULL)
          fatal("cannot open input file");
     if ((ofp = fopen(argv[2], "w")) == NULL)
          fatal("cannot open output file");
```

```
        while ((n = fread(buf, 1, BUFSIZ, ifp)) != 0) {
                if (fwrite(buf, 1, n, ofp) != n)
                        fatal("write error");
        }

        if (!feof(ifp))
                fatal("read error");
        if (fclose(ifp) == EOF || fclose(ofp) == EOF)
                fatal("cannot close");
        exit(0);
}
```

See Also

fclose(), fdopen(), freopen(), STDIO

Diagnostics

fopen returns NULL if it cannot allocate a **FILE** structure, if the *type* string is nonsense, or if the call to **open** or **creat** fails. Currently, only 20 **FILE** structures can be allocated per program, including **stdin**, **stdout**, and **stderr**.

Notes

Many operating systems recognize a 'b' modifier to the *type* argument; this indicates that the file contains binary information, and lets the operating system handle "funny characters" correctly. COHERENT has no need of such a modifier, so if you append 'b' to *type*, it will be ignored. This modifier, however, is recognized by numerous other operating systems, including MS-DOS, OS/2, and GEMDOS. If you expect to port developed code to any of these operating systems, files should append the 'b' to *type*.

for — C Keyword

Control a loop

for(*initialization*; *endcondition*; *modification*)

for is a C keyword that introduces a loop. It takes three arguments, which are separated by semicolons ';'. *initialization* is executed before the loop begins. *endcondition* describes the condition that ends the loop. *modification* is a statement that modifies *variable* to control the number of iterations of the loop. For example,

```
for (i=0; i<10; i++)
```

first sets the variable **i** to zero; then it declares that the loop will continue as long as **i** remains less than ten; and finally, increments **i** by one after every iteration of the loop. This ensures that the loop will iterate exactly ten times (from **i==0** through **i==9**). The statement

```
for(;;)
```

will loop until its execution is interrupted by a **break**, **goto**, or **return** statement. Also, either or both of *initialization* and *modification* may consist of multiple statements that are separated by commas. For example,

```
for (i=0, j=0; i<10; i++, j++)
```

initializes both *i* and *j*, and increments both with each iteration of the loop.

See Also

break, C keywords, continue, while

for — Command

Execute commands for tokens in list
for *name* [**in** *token* ...] **do** *sequence* **done**

The shell command **for** controls a loop. It assigns to the variable *name* each successive *token* in the list, and then executes the commands in the given *sequence*. If the **in** clause is omitted, **for** successively assigns *name* the value of each positional parameter to the current script ('$@'). Because the shell recognizes a reserved word only as the unquoted first word of a command, both **do** and **done** must either occur unquoted at the start of a command or be preceded by '".

The shell commands **break** and **continue** may be used to alter control flow within a **for** loop.

The shell executes **for** directly.

See Also

break, commands, continue, ksh, sh

fork() — COHERENT System Call

Create a new process
fork()

In the COHERENT system, many processes may be active simultaneously. **fork** creates a new process; the new process is a duplicate of the requesting process. In practice, the new process often issues a call to execute yet another new program.

The process that issues the **fork** call is termed the *parent* process, and the newly forked process is termed the *child* process. **fork** returns the process id of the newly created child to the parent process, and returns zero to the child process. The parent may call **wait** to suspend itself until the child terminates.

The following parts of the environment of a process are exactly duplicated by a **fork** call:

- Open files and their seek positions
- Current working and root directories
- The file creation mask
- The values of all signals
- The alarm clock setting
- Code, data, and stack segments

The system normally makes a fresh copy of the code, data, and stack segments for the child process. One advantage of *shared text* processes is that they do not need to copy the code segment. It is write protected, and therefore may be shared.

Example

For an example of how to use this call, see **pipe**.

See Also

alarm(), COHERENT system calls, execl(), exit(), sh, umask(), wait()

Diagnostics

fork returns -1 on failure, which usually involves insufficient system resources. On successful calls, **fork** returns zero to the child and the process id of the child to the parent.

fortune — Command

Print randomly selected, hopefully humorous, text
/usr/games/fortune [*file*]

fortune prints a message that is randomly selected from the contents of a text file. **fortune** reads *file* if it is named on the command line; otherwise, it reads the default file **/usr/games/lib/fortunes**.

Files

/usr/games/lib/fortunes — Default fortunes

See Also

commands

fperr.h — Header File

Constants used with floating-point exception codes
#include <fperr.h>

fperr.h declares constants used by routines that handle floating-point exceptions. It also defines the error messages they use.

See Also

header files

fprintf() — STDIO (libc)

Print formatted output into file stream
int fprintf(*fp, format,* [*arg1, ... argN*])
FILE ****fp*; char ****format***;**
[*data type*] *arg1, ... argN*;

fprintf formats and prints a string. It resembles the function **printf**, except that it writes its output into the stream pointed to by **fp**, instead of to the standard output.

fprintf uses the *format* to specify an output format for *arg1* through *argN*.

See **printf** for a description of **fprintf**'s formatting codes.

Example

For an example of this routine, see the entry for **fscanf**.

See Also

printf(), sprintf(), STDIO

Notes

Because C does not perform type checking, it is essential that an argument match its specification. For example, if the argument is a **long** and the specification is for an **int**, **fprintf** will peel off the first word of that **long** and present it as an **int**.

At present, **fprintf** does not return a meaningful value.

fputc() — STDIO (libc)

Write character into file stream
#include <stdio.h>
int fputc(*c*, *fp*) **char** *c*; **FILE** **fp*;

fputc writes the character *c* into the file stream pointed to by *fp*. It returns *c* if *c* was written successfully.

Example

The following example uses **fputc** to write the contents of one file into another.

```
#include <stdio.h>

void fatal(message)
char *message;
{
     fprintf(stderr, "%s\n", message);
     exit(1);
}

main()
{
     FILE *fp, *fout;
     int ch;
     int infile[20];
     int outfile[20];

     printf("Enter name to copy: ");
     gets(infile);
     printf("Enter name of new file: ");
     gets(outfile);

     if ((fp = fopen(infile, "r")) == NULL)
          fatal("Cannot write input file");

     if ((fout = fopen(outfile, "w")) != NULL)
          fatal("Cannot write output file");

     while ((ch = fgetc(fp)) != EOF)
          fputc(ch, fout);
}
```

See Also

STDIO

Diagnostics

fputc returns EOF when a write error occurs, e.g., when a disk runs out of space.

fputs() — STDIO (libc)

Write string into file stream
#include <stdio.h>
int fputs(*string*, *fp*) **char** **string*; **FILE** **fp*;

fputs writes *string* into the file stream pointed to by *fp*. Unlike its cousin **puts**, it does not append a newline character to the end of *string*.

fputs returns a nonnegative value on success and **EOF** if a write error occurs.

Example

For an example of this function, see the entry for **freopen**.

See Also

puts(), **STDIO**

fputw() — STDIO (libc)

Write an integer into a stream
#include <stdio.h>
int fputw(*word*, *fp***) int** *word*; **FILE** **fp*;

fputw writes *word* into the file stream pointed to by *fp*, and returns the value written.

Example

For an example of this function, see the entry for **fgetw**.

See Also

fgetw(), **STDIO**

Diagnostics

fputw returns **EOF** when an error occurs. A call to **ferror** or **feof** may be needed to distinguish this value from a valid end-of-file signal.

fread() — STDIO Function (libc)

Read data from file stream
#include <stdio.h>
int fread(*buffer, size, n, fp***)**
char **buffer*; **unsigned** *size, n*; **FILE** **fp*;

fread reads *n* items, each being *size* bytes long, from file stream *fp* into *buffer*.

Example

For an example of how to use this function, see the entry for **fopen**.

See Also

fwrite(), **STDIO**

Diagnostics

fread returns zero upon reading EOF or on error; otherwise, it returns the number of items read.

free() — General Function (libc)

Return dynamic memory to free memory pool
void free(*ptr***) char** **ptr*;

free helps you manage the arena. It returns to the free memory pool memory that had previously been allocated by **malloc**, **calloc**, or **realloc**. **free** marks the block indicated by *ptr* as unused, so the **malloc** search can coalesce it with contiguous free blocks. *ptr* must have been obtained from **malloc**, **calloc**, or **realloc**.

Example

For an example of how to use this routine, see the entry for **malloc**.

See Also

arena, calloc(), general functions, malloc(), realloc(), setbuf()

Diagnostics

free prints a message and calls **abort** if it discovers that the arena has been corrupted. This most often occurs by storing data beyond the bounds of an allocated block.

freopen() — STDIO Function (libc)

Open file stream for standard I/O
#include <stdio.h>
FILE *freopen (*name*, *type*, *fp*)
char **name*, **type*; **FILE** **fp*;

freopen reinitializes the file stream *fp*. It closes the file currently associated with it, opens or creates the file *name*, and returns a pointer to the structure for use by other STDIO routines. *name* names a file.

type is a string that consists of one or more of the characters "**rwa**" (for, respectively, read, write, and append) to indicate the mode of the stream. For further discussion of the *type* variable, see the entry for **fopen**. **freopen** differs from **fopen** only in that *fp* specifies the stream to be used. Any stream previously associated with *fp* is closed by **fclose**. **freopen** is usually used to change the meaning of **stdin**, **stdout**, or **stderr**.

Example

This example, called **match.c**, looks in **argv[2]** for the pattern given by **argv[1]**. If the pattern is found, the line that contains the pattern is written into the file **argv[3]** or to **stdout**.

```
#include <stdio.h>
#define MAXLINE 128
char buffer[MAXLINE];

void fatal(message)
char *message;
{
     fprintf(stderr, "match: %s\n", message);
     exit(1);
}

main(argc,argv)
int argc; char *argv[];
{
     FILE *fpin, *fpout;

     if (argc != 3 && argc != 4)
          fatal("Usage: match pattern infile [outfile]");
     if ((fpin = fopen(argv[2], "r")) == NULL)
          fatal("Cannot open input file");
```

```
        fpout = stdout;
        if (argc == 4)
                if ((fpout = freopen(argv[3], "w", stdout)) == NULL)
                        fatal("Cannot open output file");

        while (fgets(buffer, MAXLINE, fpin) != NULL) {
                if (pnmatch(buffer, argv[1], 1))
                        fputs(buffer, stdout);
        }
        exit(0);
}
```

See Also

fopen(), STDIO

Diagnostics

freopen returns NULL if the *type* string is nonsense or if the file cannot be opened. Currently, only 20 **FILE** structures can be allocated per program, including **stdin**, **stdout**, and **stderr**.

frexp() — General Function (libc)

Separate fraction and exponent
double frexp(*real*, *ep***) double** *real*; **int** **ep*;

frexp breaks double-precision floating point numbers into fraction and exponent. It returns the fraction *m* of its *real* argument, such that 0.5 <= *m* < 1 or *m*=0, and stores the binary exponent *e* in the location pointed to by *ep*. These numbers satisfy the equation *real* = *m* * 2*e*.

Example

This example prompts for a number, then uses **frexp** to break it into its fraction and exponent.

```
#include <stdio.h>

main()
{
        extern char *gets();
        extern double frexp(), atof();
        double real, fraction;
        int ep;

        char string[64];

        for (;;) {
                printf("Enter number: ");
                if (gets(string) == NULL)
                        break;
```

```
            fraction = frexp(real, &ep);
            printf("%lf is the fraction of %lf\n",
                 fraction, real);
            printf("%d is the binary exponent of %lf\n",
                 ep, real);
        }

        putchar('\n');
    }
```

See Also

atof(), ceil(), fabs(), floor(), general functions, ldexp(), modf()

from — Command

Generate list of numbers, for use in loop
from *start* **to** *stop* [**by** *incr*]

from prints a list of integers on the standard output, one per line. It prints beginning with *start*, and then prints successive numbers incrementing by *incr* (default, one) the previous number. It continues until the generated value matches or exceeds *stop*. Each of *start*, *stop*, and optional *incr* is a decimal integer with an optional leading '-' sign.

Typical uses of **from** include generating a file of numbers and generating a loop index for the shell. The following example creates special files for eight terminal ports:

```
for i in 'from 0 to 7'
do
     /etc/mknod /dev/hs0$i c 7 $i
done
```

See Also

commands, ksh, sh

Diagnostics

from prints an error message if the generated list is empty.

fscanf() — STDIO (libc)

Format input from a file stream
#include <stdio.h>
int fscanf(*fp, format, arg1, ... argN***)**
FILE ****fp*; char ****format***;**
[*data type*] ******arg1*, ... ******argN***;**

fscanf reads the file stream pointed to by *fp*, and uses the string *format* to format the arguments *arg1* through *argN*, each of which must point to a variable of the appropriate data type.

fscanf returns either the number of arguments matched, or EOF if no arguments matched.

For more information on **fscanf**'s conversion codes, see **scanf**.

Example

The following example uses **fprintf** to write some data into a file, and then reads it back using **fscanf**.

```
#include <stdio.h>

main ()
{
      FILE *fp;
      char let[4];

      /* open file into write/read mode */
      if ((fp = fopen("tmpfile", "wr")) == NULL) {
            printf("Cannot open 'tmpfile'\n");
            exit(1);
      }

      /* write a string of chars into file */
      fprintf(fp, "1234");

      /* move file pointer back to beginning of file */
      rewind(fp);

      /* read and print data from file */
      fscanf(fp, "%c %c %c %c",
            &let[0], &let[1], &let[2], &let[3]);
      printf("%c %c %c %c\n",
            let[3], let[2], let[1], let[0]);
}
```

See Also

scanf(), sscanf(), STDIO

Notes

Because C does not perform type checking, it is essential that an argument match its specification. For that reason, **fscanf** is best used only to process data that you are certain are in the correct data format, such as data previously written out with **fprintf**.

fsck — Command

Check and repair file systems interactively

/etc/fsck [**-fnqsSy**] [**-t** *tempfile*] [*filesystem* ...]

fsck checks and interactively repairs file systems. If all is well, **fsck** merely prints the number of files used, the number of blocks used, and the number of blocks that are free. If the file system is found to be inconsistent in one of the aspects outlined below, **fsck** asks whether it should fix the inconsistency and waits for you to reply **yes** or **no**.

The following file system aspects are checked for consistency by **fsck**:

- If a block is claimed by more than one i-node, by an i-node and the free list, or more than once in the free list.
- Whether an i-node or the free list claims blocks beyond the file system's range.
- Link counts that are incorrect.
- Whether the directory size is not aligned for 16 bytes.
- Whether the i-node format is correct.

- Whether any blocks are not accounted for.
- Whether a file points to an unallocated i-node.
- Whether a file's i-node number is out of range.
- Whether the super block refers to more than 65,536 i-nodes.
- Whether the super block assigned more blocks to the i-nodes than the system contains.
- Whether the format of the free block list is correct.
- Whether the counts of the total free blocks and the free i-nodes are correct.

fsck prints a warning message when a file name is null, has an embedded slash '/', is not null-padded, or if '.' or '..' files do not have the correct i-node numbers.

When **fsck** repairs a file system, any file that is orphaned (that is, allocated but not referenced) is deleted if it is empty, or copied to a directory called **lost+found**, with its i-node number as its name. The directory **lost+found** must exist in the root of the file system being checked before **fsck** is executed, and it must have room for new entries without requiring that new blocks be allocated.

fsck recognizes the following options:

-f Fast check. **fsck** only checks whether a block has been claimed by more than one i-node, by an i-node and the free list, or more than once in the free list. If necessary, **fsck** will reconstruct the free list.

-n No option: a default reply of **no** is given to all of **fsck**'s questions.

-q Quiet option: run quietly. **fsck** automatically removes all unreferenced pipes, and automatically fixes list counts in the super block and the free list. File-name warning messages are suppressed, but **fsck** still prints the number of files used, the number of blocks used, and the number of blocks that remain free.

-s Sort the free lists, both free blocks and free i-nodes, based on the interleave number. This is useful in reducing fragmentation of a file system. This option ignores mounted file systems.

-S Same as **-s**, except that it also works on mounted file systems. Not recommended for the faint of heart.

-t Specify temporary file option: **fsck** uses RAM device **/dev/rram1** for temporary storage when checking filesystems larger than approximately 35 megabytes. This option allows the user to specify temporary storage other than the RAM device.

-y Yes option: a default reply of **yes** is given to all of **fsck**'s questions.

If you do not name a file system in **fsck**'s command line, **fsck** checks the file systems named in the file **/etc/checklist**.

If **fsck** is invoked for a file system larger than approximately 35 megabytes, it uses the RAM device **/dev/rram1** for temporary storage. For this reason, it is strongly advised that you not use **/dev/rram1** as a RAM disk.

Files

/etc/checklist

See Also

clri, commands, icheck, ncheck, ram, sync, umount

Notes

The correction of file systems almost always involves the destruction of data.

You should run **fsck** only when the COHERENT system is in single-user mode.

Previous editions of **fsck** could check no partition larger than 35 megabytes. This restriction has been lifted.

fseek() — STDIO Function (libc)

Seek on file stream
#include <stdio.h>
int fseek(*fp*, *where*, *how*)
FILE **fp*; **long** *where*; **int** *how*;

fseek changes where the next read or write operation will occur within the file stream *fp*. It handles any effects the seek routine might have had on the internal buffering strategies of the system. The arguments *where* and *how* specify the desired seek position. *where* indicates the new seek position in the file. It is measured from the start of the file if *how* equals zero, from the current seek position if *how* equals one, and from the end of the file if *how* equals two.

fseek differs from its cousin **lseek** in that **lseek** is a COHERENT system call and takes a file number, whereas **fseek** is a STDIO function and takes a **FILE** pointer.

Example

This example opens file **argv[1]** and prints its last **argv[2]** characters (default, 100). It demonstrates the functions **fseek**, **ftell**, and **fclose**.

```
#include <stdio.h>
extern long atol();

void fatal(message)
char *message;
{
     fprintf(stderr, "tail: %s\n", message);
     exit(1);
}

main(argc, argv)
int argc; char *argv[];
{
     register FILE *ifp;
     register int c;
     long nchars, size;

     if (argc < 2 || argc > 3)
          fatal("Usage: tail file [ nchars ]");
     nchars = (argc == 3) ? atol(argv[2]) : 100L;
```

```
        if ((ifp = fopen(argv[1], "r")) == NULL)
             fatal("cannot open input file");
        /* Seek to end */
        if (fseek(ifp, 0L, 2) == -1)
             fatal("seek error");

        /* Find current size */
        size = ftell(ifp);
        size = (size < nchars) ? 0L : size - nchars;

        /* Seek to point */
        if (fseek(ifp, size, 0) == -1)
             fatal("seek error");
        while ((c = getc(ifp)) != EOF)
             /* Copy rest to stdout */
             putchar(c);
        if (fclose(ifp) == EOF)
             fatal("cannot close");
        exit(0);
}
```

See Also

ftell(), lseek(), STDIO

Diagnostics

For any diagnostic error, **fseek** returns -1; otherwise, it returns zero. If **fseek** goes beyond the end of the file, it will not return an error message until the corresponding read or write is performed.

fstat() — General Function (libc)

Find file attributes

#include <sys/stat.h>

fstat(*descriptor, statptr***) int** *descriptor*; **struct stat** **statptr*;

fstat returns a structure that contains the attributes of a file including protection information, file type, and file size. *descriptor* is the file descriptor for the open file, and *statptr* points to a structure of the type **stat**, which is defined in the header file **stat.h**.

The following summarizes the structure **stat** and defines the permission and file type bits.

```
struct stat {
      dev_t st_dev;
      ino_t st_ino;
      unsigned short st_mode;
      short st_nlink;
      short st_uid;
      short st_gid;
      dev_t st_rdev;
      size_t st_size;
      time_t st_atime;
      time_t st_mtime;
      time_t st_ctime;
};

#define S_IFMT 0170000        /* file types */
#define S_IFREG 0100000       /* ordinary file */
#define S_IFDIR 0040000       /* directory */
#define S_IFCHR 0020000       /* character special */
#define S_IFBLK 0060000       /* block special */
#define S_ISUID 0004000       /* set user id */
#define S_ISGID 0002000       /* set group id */
#define S_ISVTX 0001000       /* save text bit */
#define S_IREAD 0000400       /* owner read permission */
#define S_IWRITE 000200       /* owner write permission */
#define S_IEXEC 0000100       /* owner execute permission */
```

The entries **st_dev** and **st_ino** together form a unique description of the file. The former is the device on which the file and its i-node reside, whereas the latter is the index number of the file. The entry **st_mode** gives the permission bits, as outlined above. The entry **st_nlink** gives the number of links to the file. The user id and group id of the owner are **st_uid** and **st_gid**, respectively. The entry **st_rdev**, valid only for special files, holds the major and minor numbers for the file.

The entry **st_size** gives the size of the file, in bytes. For a pipe, the size is the number of bytes waiting to be read from the pipe.

Three entries for each file give the last occurrences of various events in the file's history. **st_atime** gives time the file was last read or written to. **st_mtime** gives the time of the last modification, write for files, create or delete entry for directories. **st_ctime** gives the last change to the attributes, not including times and size.

Example

For an example of how to use this function, see the entry for **pipe**.

Files

<sys/stat.h>

See Also

chmod(), chown(), COHERENT system calls, ls, open(), stat()

Notes

fstat differs from the related function **stat** mainly in that it accesses the file through its descriptor, which was returned by a successful call to **open**, whereas **stat** takes the file's path name and opens it itself before checking its status.

Diagnostics

fstat returns -1 if the file is not found or if *statptr* is invalid.

ftell() — STDIO Function (libc)

Return current position of file pointer
#include <stdio.h>
long ftell(*fp***) FILE ***fp***;**

ftell returns the current position of the seek pointer. Like its cousin **fseek**, **ftell** takes into account any buffering that is associated with the stream *fp*.

Example

For an example of how to use this function, see the entry for **fseek**.

See Also

fseek(), lseek(), rewind(), STDIO

ftime() — Time Function (libc)

Get the current time from the operating system
#include <sys/timeb.h>
ftime(*tbp***) struct timeb ***tbp***;**

ftime fills the structure **timeb**, which is pointed to *tbp*, with COHERENT's representation of the current time. **timeb** is defined in the header file **timeb.h**, as follows:

```
struct timeb {
     time_t time;
     unsigned short millitm;
     short timezone;
     short dstflag;
}
```

The member **time** is the number of seconds since January 1, 1970, 0h00m00s GMT. **millitm** is a count of milliseconds. **timezone** and **dstflag** are obsolete; they have been replaced by the environmental variable **TIMEZONE**.

See Also

date, time, TIMEZONE, types.h

function — Definition

A **function** is the C term for a portion of code that is named, can be invoked by name, and that performs a task. Many functions can accept data in the form of arguments, modify the data, and return a value to the statement that invoked it.

See Also

data types, definitions, executable file, library, portability

fwrite() — STDIO Function (libc)

Write into file stream
#include <stdio.h>
int fwrite(*buffer, size, n, fp*)
char **buffer*; **unsigned** *size, n*; **FILE** **fp*;

fwrite writes *n* items, each of *size* bytes, from *buffer* into the file stream pointed to by *fp*.

Example

For an example of how to use this function, see the entry for **fopen**.

See Also

fread(), STDIO

Diagnostics

fwrite normally returns the number of items written. If an error occurs, the returned value will not be the same as *n*.

fwtable — Command

Build font-width table
fwtable [**-pv**] [*infile* [*outfile*]]

fwtable builds a binary font-width table for use by **troff**. It understands PCL (Printer Control Language) bitmap fonts for the Hewlett-Packard LaserJet family of printers (plus compatibles), and AFM (Adobe Font Metric) descriptions of PostScript fonts.

For the typesetting program **troff** to use a font, it must know the width of each character in the font and how to tell the printer to select the font. **troff** contains built-in information about a few standard fonts, but to use any other font you must use the **troff** directive **.lf** to load a binary *font-width table* that contains information about the font. The command **fwtable** normally reads a PCL bitmap font for an HP-compatible laser printer from *infile* (or the standard input) and writes a font-width table for the font to *outfile* (or the standard output).

Loading a PCL **troff** font-width table with an **.lf** directive provides **troff** with character-width information about the font and tells it the PCL command (escape sequence) required to select the font. However, it does *not* download the font to the printer. You must download each required font to the printer with the **hpr** command (using its **-f** option) before you print the **troff** output; if the fonts are not available in the printer, the output will not be what you expect.

With option **-p**, **fwtable** reads an AFM (Adobe Font Metric) description for a PostScript font from *infile* and writes a font-width table to *outfile*.

With option **-v**, **fwtable** prints a brief font description to the standard error file.

Files

/usr/lib/roff/troff_pcl/fwt/ — Directory for PCL font-width tables
/usr/lib/roff/troff_ps/fwt/ — Directory for PostScript font-width tables

See Also

commands, hpr, troff

Notes

fwtable does not understand Intellifont scalable fonts.

G

gcd() — Multiple-Precision Mathematics

Set variable to greatest common divisor

#include <mprec.h>
void gcd(*a, b, c***)**
mint ****a*, ******b*, ***c;

The COHERENT system includes a suite of routines that allow you to perform multiple-precision mathematics. **gcd** sets *c* to the greatest common divisor of *a* and *b*.

See Also

multiple-precision mathematics

general functions — Overview

The library **libc** includes a number of functions that perform useful, general tasks:

abort() . . . . . End program immediately
abs(). . . . . Return the absolute value of an integer
assert() . . . . . Check assertion at run time
atof() . . . . . Convert ASCII strings to floating point
atoi() . . . . . Convert ASCII strings to integers
atol() . . . . . Convert ASCII strings to long integers
bsearch(). . . . . Search an array
calloc() . . . . . Allocate dynamic memory
candaddr(). . . . . Convert a **daddr_t** to canonical format
candev() . . . . . Convert a **dev_t** to canonical format
canino() . . . . . Convert a **ino_t** to canonical format
canint(). . . . . Convert a **int** to canonical format
canlong(). . . . . Convert a **long** to canonical format
canshort() . . . . . Convert a **short** to canonical format
cansize() . . . . . Convert an **fsize_t** to canonical format
cantime(). . . . . Convert a **time_t** to canonical format
canvaddr() . . . . . Convert a **vaddr_t** to canonical format
crypt() . . . . . Encryption using rotor algorithm
div(). . . . . Perform integer division
endgrent() . . . . . Close group file
endpwent() . . . . . Close password file
exit() . . . . . Terminate a program
free() . . . . . Return dynamic memory to free memory pool
frexp() . . . . . Separate fraction and exponent
getenv() . . . . . Read environmental variable
getgrent() . . . . . Get group file information
getgrgid(). . . . . Get group file information, by group id
getgrnam(). . . . . Get group file information, by group name
getlogin(). . . . . Get login name
getopt(). . . . . Get a command-line option
getpass() . . . . . Get password with prompting
getpw() . . . . . Search password file
getpwent() . . . . . Get password file information
getpwnam() . . . . . Get password file information, by name
getpwuid() . . . . . Get password file information, by id
getwd() . . . . . Get current working directory name
isatty() . . . . . Check if a device is a terminal

l3tol() . . . . . Convert file system block number to long integer
ldexp() . . . . . Combine fraction and exponent
ldiv() . . . . . Perform long integer division
longjmp(). . . . . Return from a non-local goto
ltol3() . . . . . Convert long integer to file system block number
malloc() . . . . . Allocate dynamic memory
mktemp() . . . . . Generate a temporary file name
modf() . . . . . Separate integral part and fraction
mtype(). . . . . Return symbolic machine type
nlist() . . . . . Symbol table lookup
path() . . . . . Build a path name for a file
perror() . . . . . System call error messages
qsort() . . . . . Sort arrays in memory
rand() . . . . . Generate pseudo-random numbers
realloc() . . . . . Reallocate dynamic memory
setgrent() . . . . . Rewind group file
setjmp() . . . . . Perform non-local goto
setpwent() . . . . . Rewind password file
shellsort() . . . . . Sort arrays in memory
sleep() . . . . . Suspend execution
srand() . . . . . Seed random number generator
strtod() . . . . . Convert string to floating-point number
strtol() . . . . . Convert string to long integer
strtoul() . . . . . Convert string to unsigned long integer
swab(). . . . . Swap a pair of bytes
system() . . . . . Pass a command to the shell for execution
ttyname() . . . . . Identify a terminal
ttyslot() . . . . . Return a terminal's line number

See Also

libraries

getc() — STDIO Macro (stdio.h)

Read character from file stream
#include <stdio.h>
int getc(*fp***) FILE ***fp***;**

getc is a macro that reads a character from the file stream *fp*, and returns an **int**.

Example

The following example creates a simple copy utility. It opens the first file named on the command line and copies its contents into the second file named on the command line.

```
#include <stdio.h>

void fatal(string)
char *string;
{
     printf("%s\n", string);
     exit (1);
}
```

```
main(argc, argv)
int argc; char *argv[];
{
     int foo;
     FILE *source, *dest;

     if (--argc != 2)
          fatal("Usage: copy [source][destination]");

     if ((source = fopen(argv[1], "r")) == NULL)
          fatal("Cannot open source file");
     if ((dest = fopen(argv[2], "w")) == NULL)
          fatal("Cannot open destination file");

     while ((foo = getc(source)) != EOF)
          putc(foo, dest);
}
```

See Also

fgetc(), getchar(), putc(), STDIO

Diagnostics

getc returns EOF at end of file or on read fatal.

Notes

Because **getc** is a macro, arguments with side effects probably will not work as expected. Also, because **getc** is a complex macro, its use in expressions of too great a complexity may cause unforeseen difficulties. Use of the function **fgetc** may avoid this.

getchar() — STDIO Macro (stdio.h)

Read character from standard input
#include <stdio.h>
int getchar()

getchar is a macro that reads a character from the standard input. It is equivalent to **getc(stdin)**.

Example

The following example gets one or more characters from the keyboard, and echoes them on the screen.

```
#include <stdio.h>

main()
{
     int foo;
     while ((foo = getchar()) != EOF)
          putchar(foo);
}
```

See Also

getc(), putchar(), STDIO

Diagnostics

getchar returns **EOF** at end of file or on read error.

getegid() — COHERENT System Call

Get effective group identifier
getegid()

Every process has two different versions of its *group identifier*, called the *real* group identifier and the *effective* group identifier. The group identifiers determine eligibility to access files and use system privileges. Normally, these two identifiers are identical. However, for a *set group identifier* load module (see **exec**), the real group identifier is that of the group's current group, whereas the effective group identifier is that of the load module owner. This distinction allows system programs to use files which are protected from groups that invoke the program.

getegid returns the effective group identifier.

See Also

access, COHERENT system calls, exec, geteuid(), getgid(), getuid(), login, setuid()

getenv() — General Function (libc)

Read environmental variable
char *getenv(*VARIABLE***) char ***VARIABLE***;**

A program may read variables from its *environment*. This allows the program to accept information that is specific to it. The environment consists of an array of strings, each having the form *VARIABLE=VALUE*. When called with the string *VARIABLE*, **getenv** reads the environment, and returns a pointer to the string *VALUE*.

Example

This example prints the environmental variable **PATH**.

```
#include <stdio.h>

main()
{
	char *env;
	extern char *getenv();

	if ((env = getenv("PATH")) == NULL) {
		printf("Sorry, cannot find PATH\n");
		exit(1);
	}
	printf("PATH = %s\n", env);
}
```

See Also

environmental variables, envp, exec, sh

Diagnostics

When *VARIABLE* is not found or has no value, **getenv** returns NULL.

geteuid() — COHERENT System Call

Get effective user identifier
geteuid()

Every process has two different versions of its *user id*, called the *real* user id and the *effective* user id. The user ids determine eligibility to access files or employ system privileges. Normally, these two ids are identical. However, for a *set user id* load module (see **exec**), the real user id is that of the user, whereas the effective user id is that of the load module owner. This distinction allows system programs to use files which are protected from the user who invokes the program.

geteuid returns the effective user identifier

See Also

access(), COHERENT system calls, exec, getegid(), getgid(), getuid(), login, setuid()

getgid() — COHERENT System Call

Get real group identifier
getgid()

Every process has two different versions of its *user id*, called the *real* user id and the *effective* user id. The user ids determine eligibility to access files or employ system privileges. Normally, these two ids are identical. However, for a *set user id* load module (see **exec**), the real user id is that of the user, whereas the effective user id is that of the load module owner. This distinction allows system programs to use files which are protected from the user who invokes the program.

getgid returns the real group id.

See Also

access(), COHERENT system calls, exec, getegid(), geteuid(), getuid(), login, setuid()

getgrent() — General Function (libc)

Get group file information
#include <grp.h>
struct group *getgrent();

getgrent returns the next entry from file **/etc/group**. It returns NULL if an error occurs or if the end of file is encountered.

Files

/etc/group
<grp.h>

See Also

general functions, group

Notes

All structures and information returned are in a static area internal to **getgrent**. Therefore, information from a previous call is overwritten by each subsequent call.

getgrgid() — General Function (libc)

Get group file information, by group name
#include <grp.h>
struct group *getgrgid(*gid***);**
int *gid*;

getgrgid searches file **/etc/group** for the first entry with a numerical group id of *gid*. It returns a pointer to the entry if found; it returns NULL if an error occurs or if the end of file is encountered.

Files

/etc/group
<grp.h>

See Also

general functions, group

Notes

All structures and information returned are in a static area internal to **getgrgid**. Therefore, information from a previous call is overwritten by each subsequent call.

getgrnam() — General Function (libc)

Get group file information, by group id
#include <grp.h>
struct group *getgrnam(*gname***);**
char *gname*;

getgrnam searches file **/etc/group** for the first entry with a group name of *gname*. It returns a pointer to the entry for *gname* if it is found; it returns NULL for any error or if the end of the file is encountered.

Files

/etc/group
<grp.h>

See Also

general functions, group

Notes

All structures and information returned are in a static area internal to **getgrnam**. Therefore, information from a previous call is overwritten by each subsequent call.

getlogin() — General Function (libc)

Get login name
char *getlogin()

The name corresponding to the current user id is not always the same as the name under which a user logged into the COHERENT system. For example, the user may have issued a **su** command, or there may be several login names associated with a user id. **getlogin** returns the login name found in the file **/etc/utmp**.

In cases where **getlogin** fails to produce a result, **getpwuid** (described in **getpwent**) is normally used to determine the user name for a process.

Files

/etc/utmp login names

See Also

general functions, getpwent(), getuid(), su, ttyname(), utmp.h, who

Diagnostics

getlogin returns NULL if the login name cannot be determined.

Notes

getlogin stores the returned name in a static area that is destroyed by subsequent calls.

getopt() — General Function (libc)

Get option letter from argv

int getopt(*argc, argv, optstring***)**
int *argc*;
**char ****argv*;
**char ***optstring*;
**extern char ***optarg*;
extern int *optind*;

getopt returns the next option letter in *argv* that matches a letter in *optstring*. *optstring* is a string of recognized option letters. If a letter is followed by a colon, the option is expected to have an argument, which may or may not be separated from it by white space. *optarg* is set to point to the start of the option argument on return from *getopt*.

getopt places into *optind* the *argv* index of the next argument to be processed. Because *optind* is external, it is normally initialized to one automatically before the first call to *getopt*.

When all options have been processed (i.e., up to the first non-option argument), *getopt* returns EOF. The special option "--" may be used to delimit the end of the options: **getopt** will return EOF and skip "--".

Example

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options **a** and **b**, and the options **f** and **o**, both of which require arguments:

```
main(argc, argv)
int argc;
char **argv;
{
     int c;
     extern int optind;
     extern char *optarg;

     .
     .
     .

     while ((c = getopt(argc, argv, "abf:o:")) != EOF)
          switch (c) {
```

```
            case 'a':
                 if (bflg)
                      errflg++;
                 else
                      aflg++;
                 break;

            case 'b':
                 if (aflg)
                      errflg++;
                 else
                      bflg++;
                 break;

            case 'f':
                 ifile = optarg;
                 break;

            case 'o':
                 ofile = optarg;
                 break;

            case '?':
            default:
                 errflg++;
                 break;
            }

      if (errflg) {
            fprintf(stderr, "Usage: ...");
            exit(2);
      }

      for (; optind < argc; optind++) {
            .
            .
            .
      }
      .
      .
      .
}
```

See Also

general functions

Diagnostics

getopt prints an error message on *stderr* and returns a question mark when it encounters an option letter not included in *optstring*.

Notes

It is not obvious how '-' standing alone should be treated. This version treats it as a non-option argument, which is not always right.

Option arguments are allowed to begin with '-'. This is reasonable, but reduces the amount of error checking possible.

getopt returns the parsed letter option in the external **int optopt**, which is overwritten by each call to **getopt**. When **getopt** returns '?', it can be helpful to examine the contents of this variable.

getopts — Command

Parse command-line options
getopts *optstring name [opt]*

The command **getopts** is available under the Korn **ksh** to parse a command's options and check their legality. *optstring* must contain the options letters that the command using **getopts** will recognize. If a letter is followed by a colon ':', that option must have an argument that is separated from it by whitespace.

Each time it is invoked, **getopts** places the next option into the shell variable *name* and the index of the next argument to be processed into the shell variable **OPTIND**, which is initialized by default to one. When an option requires an argument, **getopts** copies it into the shell variable **OPTARG**. If **getopts** encounters an error, it initializes variable *name* to **?**.

When it encounters the end of the options, **getopts** exits with non-zero status. The special option "--" can be used to delineate the end of options.

Example

The following example processes a command that takes options **a**, **b**, and **o**; the last option requires an argument:

```
while getopts abo: c
do
     case $c in
          a|b) FLAGS=$FLAGS$c;;
          o)   OARG=$OPTARG;;
          \?)  echo $USAGE 1>&2
               exit 2;;
     esac
done
shift OPTIND-1
```

This code will accept any of the following as equivalent:

```
cmd -a -b -o "xxx z yy" file
cmd -a -b -o "xxx z yy" -- file
cmd -ab -o "xxx z yy" file
cmd -ab -o "xxx z yy" -- file
```

See Also

commands, getopt(), ksh

getpass() — General Function (libc)

Get password with prompting
char *getpass(*prompt***)**
char ***prompt*;**

getpass first prints the *prompt*. Then it disables echoing of input characters on the terminal device (either the file **/dev/tty** or the standard input), reads a password from it, and restores echoing on the terminal. It returns the given password.

Files

/dev/tty

See Also

crypt(), general functions, login, passwd, su

Notes

The password is stored in a static location that is overwritten by successive calls.

getpid() — COHERENT System Call

Get process identifier
getpid()

Every process has a unique number, called its *process id*. **fork** returns the process id of a created child process to the parent process.

getpid returns the process id of the requesting process. Typically a process uses **getpid** to pass its process id to another process which wants to send it a signal, or to generate a unique temporary file name.

See Also

COHERENT system calls, fork(), kill, mktemp

getpw() — General Function (libc)

Search password file
getpw(*uid, line***)**
short *uid***;**
char ***line*;**

getpw searches the password file **/etc/passwd** for the first entry with numerical user id *uid*. If found, *line* receives the corresponding line from the password file.

Files

/etc/passwd

See Also

general

Diagnostics

getpw returns a nonzero value on error.

getpwent() — General Function (libc)

Get password file information
#include <pwd.h>
struct passwd *getpwent()

The COHERENT system has five routines that search the file **/etc/passwd**, which contains information about every user of the system. The returned structure **passwd** is defined in the header file **pwd.h**. For a description of this structure, see **pwd.h**.

getpwent returns the next entry from **/etc/passwd**.

Files

/etc/passwd
<pwd.h>

See Also

endpwent(), general functions, getpwnam(), getpwuid(), pwd.h, setpwent()

Diagnostics

getpwent returns NULL for any error or on end of file.

Notes

All structures and information returned are in static areas internal to **getpwent**. Therefore, information from a previous call is overwritten by each subsequent call.

getpwnam() — General Function (libc)

Get password file information, by name
#include <pwd.h>
struct passwd *getpwnam(*uname***)**
char ***uname*;**

The COHERENT system has five routines that search the file **/etc/passwd**, which contains information about every user of the system. The returned structure **passwd** is defined in the header file **pwd.h**. For a description of this structure, see **pwd.h**.

getpwnam attempts to find the first entry with a name of *uname*.

Files

/etc/passwd
<pwd.h>

See Also

endpwent(), general functions, getpwent(), getpwuid(), pwd.h, setpwent()

Diagnostics

getpwnam returns NULL for any error or on end of file.

Notes

All structures and information returned are in static areas internal to **getpwnam**. Therefore, information from a previous call is overwritten by each subsequent call.

getpwuid() — General Function (libc)

Get password file information, by id
#include <pwd.h>
struct passwd *getpwuid(*uid***)**
int *uid***;**

The COHERENT system has five routines that search the file **/etc/passwd**, which contains information about every user of the system. The returned structure **passwd** is defined in the header file **pwd.h**. For more information on this structure, see **pwd.h**.

getpwuid attempts to find the first entry with a numerical user id of *uid*.

Files

/etc/passwd
<pwd.h>

See Also

endpwent(), general functions, getpwent(), getpwnam(), pwd.h, setpwent()

Diagnostics

getpwuid returns NULL for any error or on end of file.

Notes

All structures and information returned are in static areas internal to **getpwuid**. Therefore, information from a previous call is overwritten by each subsequent call.

gets() — STDIO Function (libc)

Read string from standard input
#include <stdio.h>
char *gets(*buffer***) char ****buffer***;**

gets reads characters from the standard input into a buffer pointed at by *buffer*. It stops reading as soon as it detects a newline character or EOF. **gets** discards the newline or EOF, appends a null character onto the string it has built, and returns another copy of *buffer*.

Example

The following example uses **gets** to get a string from the console; the string is echoed twice to demonstrate what **gets** returns.

```
#include <stdio.h>

main()
{
     char buffer[80];

     printf("Type something: ");

     /*
      * because of the way COHERENT's teletype
      * driver works, the following fflush has
      * no effect.  It should be included for
      * portability to other operating systems.
      */

     fflush(stdout);
     printf("%s\n%s\n", gets(buffer), buffer);
}
```

See Also

buffer, fgets(), getc(), STDIO

Diagnostics

gets returns NULL if an error occurs or if EOF is seen before any characters are read.

Notes

gets stops reading the input string as soon as it detects a newline character. If a previous input routine left a newline character in the standard input buffer, **gets** will read it and immediately stop accepting characters; to the user, it will appear as if **gets** is not working at all.

For example, if **getchar** is followed by **gets**, the first character **gets** will receive is the newline character left behind by **getchar**. A simple statement will remedy this:

```
while (getchar() != '\n')
        ;
```

This throws away the newline character left behind by **getchar**; **gets** will now work correctly.

getty — System Maintenance

Terminal initialization
/etc/getty *type*

The initialization process **init** invokes **getty** for each terminal indicated in the file **/etc/ttys**. **getty** tries to read a user name from the terminal which is the standard input, adapting its mode settings accordingly. Then **getty** invokes **login** with the name read. This process may set delays, mapping of upper to lower case, speed, and whether the terminal normally uses carriage return or linefeed to terminate input.

If the terminal baud rate is wrong, the login message printed by **getty** will appear garbled. If the specified *type* indicates variable speeds, as described below, hitting BREAK will try the next speed.

init passes the third character in a line of the file **/etc/ttys** as the *type* argument to **getty**. *type* conveys information about the terminal port. An upper-case letter in the range **A** to **S** specifies a hard-wired baud rate, as indicated in the header file **<sgtty.h>**. Other characters specify a range of speeds suitable to a dial-in modem. The following variable-speed settings are recognized:

0 Cycles through speeds 300, 1200, 150, and 110 baud, in that order. This is a good default setting for dial-in ports.

- Teletype model 33, fixed at 110 baud.

1 Teletype model 37, fixed at 150 baud.

2 9600 baud with delays (e.g., Tektronix 4104).

3 Cycles between 2400, 1200, and 300 baud. This is used with 2400-bps modems.

4 DECwriter (LA36) with delays.

5 Like **3**, but starts at 300 baud.

getty recognizes the following fixed-speed settings, for hard-wired terminals:

A	50 baud
B	75 baud
C	110 baud
D	134 baud
E	150 baud
F	200 baud
G	300 baud
H	600 baud

I	1200 baud
J	1800 baud
K	2000 baud
L	2400 baud
M	3600 baud
N	4800 baud
O	7200 baud
P	9600 baud
Q	19200 baud
R	EXT
S	EXT

Files

/etc/tty
<sgtty.h>

See Also

init, ioctl(), login, sgtty.h, system maintenance, stty, ttys

getuid() — COHERENT System Call

Get real user identifier
getuid()

Every process has two different versions of its *user id*, called the *real* user id and the *effective* user id. The user ids determine eligibility to access files or employ system privileges. Normally, these two ids are identical. However, for a *set user id* load module (see **exec**), the real user id is that of the user, whereas the effective user id is that of the load module owner. This distinction allows system programs to use files which are protected from the user who invokes the program.

getuid returns the real user id.

See Also

access(), COHERENT system calls, exec, getegid(), geteuid(), getgid(), login, setuid()

getw() — STDIO Function (libc)

Read word from file stream
#include <stdio.h>
int getw(*fp*) FILE **fp*;

getw reads a word (an **int**) from the file stream *fp*.

getw differs from **getc** in that **getw** gets and returns an **int**, whereas **getc** returns either a **char** promoted to an **int**, or EOF. To detect EOF while using **getw**, you must use **feof**.

See Also

canon, getc(), STDIO

Notes

getw returns EOF on errors.

getw assumes that the bytes of the word it receives are in the natural byte ordering of the machine. This means that such files might not be portable between machines.

getwd() — General Function (libc)

Get current working directory name
char *getwd(*pathname***)**
**char ***pathname*

The *current working directory* is the directory from which file name searches commence when a pathname does not begin with '/'. **getwd** returns the name of the current working directory. It is useful for processes like spoolers and daemons, which must generate full path names for files.

If you do not have permission to search all levels of the directory hierarchy above the current directory, **getwd** cannot obtain the directory name for you.

See Also

chdir(), general functions, pwd

Diagnostics

getwd returns NULL and writes an error message into *pathname* if an error occurs, e.g., if the current directory cannot be found or if any other error occurs.

Notes

getwd fails if the current directory name is longer than MAXPATHLEN characters (1,024 characters as defined in header file **<sys/param.h>**). The chunk of memory pointed to by *pathname* must be big enough to hold **MAXPATHLEN** characters plus a trailing NUL.

If **getwd** fails, the working directory cannot be restored to its initial value.

GMT — Definition

GMT is an abbreviation of Greenwich Mean Time, the time recorded at the Greenwich Observatory in England, where by international convention the Earth's zero meridian is fixed.

By definition, COHERENT fixes system time in GMT. It calculates local time as an offset of GMT; for example, the time zone for Chicago is six hours (360 minutes) behind Greenwich, so the local time for Chicago is calculated by subtracting 360 minutes from GMT.

See Also

definitions, gmtime(), localtime, time, time.h, TIMEZONE

Notes

The ANSI Standard replaces GMT with UTC (*universel temps coordonne*, or universal coordinated time) for C programming. The change is mainly one of terminology rather than substance, as some signatories to international conventions object to naming the standard for global time after a village in England.

Under international convention, there are two UTC standards: UTC1 is based on solar time and is identical to current GMT, whereas UTC2 uses atomic clocks that are corrected by comparison with pulsars. These standards drift apart as the earth's rotation slows; thus, "leap seconds" are inserted periodically into UTC1 to bridge the difference.

gmtime() — Time Function (libc)

Convert system time to calendar structure
#include <time.h>
#include <sys/types.h>
tm *gmtime(*timep***) time_t ***timep***;**

gmtime converts the internal time from seconds since midnight January 1, 1970 GMT, into fields that give integer years since 1900, the month, day of the month, the hour, the minute, the second, the day of the week, and yearday. It returns a pointer to the structure **tm**, which defines these fields, and which is itself defined in the header file **time.h**. Unlike its cousin, **localtime**, **gmtime** returns Greenwich Mean Time (GMT).

Example

For an example of how to use this function, see **asctime**.

See Also

GMT, localtime(), time, TIMEZONE

Notes

gmtime returns a pointer to a statically allocated data area that is overwritten by successive calls.

goto — C Keyword

Unconditionally jump within a function

A **goto** command jumps to the area of the program introduced by a label. A program can **goto** only within a function; to jump across function boundaries, you must use the functions **setjmp** and **longjmp**.

In the context of C programming, the most common use for **goto** is to exit from a control block or go to the top of a control block. It is used most often to write "ripcord" routines, i.e., routines that are executed when an major error occurs too deeply within a function for the program to disentangle itself correctly. Note that in most instances, **goto** is a bad solution to a problem that can be better solved by structured programming.

Example

The following example demonstrates how to use **goto**.

```
#include <stdio.h>

main()
{
     char line[80];

getline:
     printf("Enter line: ");
     fflush(stdout);
     gets(line);
```

```
/* a series of tests often is best done with goto's */
     if (*line == 'x') {
          printf("Bad line\n");
          goto getline;
     } else if (*line == 'y') {
          printf("Try again\n");
          goto getline;
     } else if (*line == 'q')
          goto goodbye;
     else
          goto getline;


goodbye:
     printf("Goodbye.\n");
     exit(0);
}
```

See Also

C keywords

Notes

The C Programming Language describes **goto** as "infinitely-abusable": *caveat utilitor.*

grep — Command

Pattern search

grep [option ...] [*pattern*] [*file* ...]

grep searches each *file* for occurrences of the *pattern* (sometimes called a regular expression). If no *file* is specified, **grep** searches the standard input. The *pattern* is given in the same manner as to **ed**. Normally, **grep** prints each line matching the *pattern.*

The following options are available.

-b With each output line, print the block number in which the line started (used to search file systems).

-c Print the count of matching lines rather than the lines.

-e The next argument is *pattern* (useful if the pattern starts with '-').

-f The next argument is a file containing a list of patterns separated by newlines; there is no *pattern* argument.

-h When more than one *file* is specified, output lines are normally accompanied by the file name; **-h** suppresses this.

-l Print the name of each file containing matching lines rather than the lines.

-n The line number in the file accompanies each line printed.

-s Suppress all output, just return status.

-v Print a line if the pattern is *not* found in the line.

-x Print the line only if it is exactly the same as the pattern; treat wildcards in the pattern as plain text.

-y Lower-case letters in the pattern match lower-case *and* upper-case letters on the input lines.

See Also

awk, cgrep, commands, ed, egrep, expr, lex, sed

Diagnostics

grep returns an exit status of zero for success, one for no matches, two for error.

Notes

cgrep is a version of **grep** that is optimized for handling C-style expressions.

egrep is an extended and faster version of **grep**.

group — File Format

Group file format

The group file **/etc/group** describes the user groups that have been defined on your COHERENT system. This allows users to control the access that members of their group have to certain files. **/etc/group** contains the information to map any ASCII group name to the corresponding numerical group identifier, and vice versa. It also contains, in ASCII, the names of the members of each group. This information is used by, among others, the command **newgrp**.

Each group has an entry in the file **/etc/group** one line per entry. Each line consists of four colon-separated ASCII fields, as follows:

group_name : *password* : *group_number* : *member*[,*member*...]

Passwords are encrypted with **crypt**, so the group file is generally readable.

The COHERENT system has five system calls that manipulate **/etc/group**, as follows:

endgrent Close **/etc/group**.

getgrent Return the next entry from **/etc/group**.

getgrnam Return the first entry with a given group name.

getgrgid Return the first entry with a given group identifier.

setgrent Rewind **/etc/group**, so that searches can begin again from the beginning of the file.

The calls **getgrent**, **getgrid**, and **getgrnam** each return a pointer to structure **group**, which is defined in the header file **grp.h** as follows:

```
struct group {
        char    *gr_name;       /* Group name */
        char    *gr_passwd;     /* Group password */
        int     gr_gid;         /* Numeric group id */
        char    **gr_mem;       /* Group members */
};
```

Files

/etc/group

See Also

chgrp(), crypt(), endgrent(), file formats, getgrent(), getgrgid(), getgrnam(), grp.h, newgrp, passwd, setgrend()

Notes

At present the group password field cannot be set directly (no command similar to **passwd** exists for groups). One alternative is to set the password in the **/etc/passwd** file for a user with the **passwd** command, and then transcribe the password into the group file manually.

grp.h — Header File

Declare group structure
#include <grp.h>

The header file **grp.h** declares the structure **group**, which is composed as follows:

```
struct group {
        char *gr_name;      /* group name */
        char *gr_passwd;    /* group password */
        int  gr_gid;        /* numeric group id */
        char **gr_mem;      /* group members */
};
```

This structure holds information about the group to which a given user belongs. It is used by the functions **endgrent**, **getgrent**, **getgrgid**, **getgrnam**, and **setgrent**.

See Also

header files

gtty() — COHERENT System Call (libc)

Device-dependent control
#include <sgtty.h>
int gtty(*fd*, *sgp***)**
int *fd***;**
struct sgttyb ***_sgp_;**

gtty gets attributes of a terminal. It is shorthand notation for **ioctl** calls with a *command* argument of **TIOCGETP**.

Example

For examples of this system call, see **pipe** and **stty()**.

Files

<sgtty.h>

See Also

COHERENT system calls, exec, ioctl(), open(), read(), stty(), write()

H

hard disk — Technical Information

The hard disk is the primary means of storing and accessing data under the COHERENT system. This article introduces some aspects of the COHERENT system that affect the care and feeding of your hard disk.

Device Drivers

The COHERENT system comes with two sets of drivers for hard disks: the **at** drivers, for AT-style hard disks; and the **scsi** drivers, for the SCSI family of hard disks. See their respective articles in the Lexicon for details.

Partitioning

The COHERENT command **fdisk** displays information about how your hard disk is currently configured. You can also use it to repartition your hard disk and reassign partitions from MS-DOS to COHERENT, or vice versa.

Note that this is an extremely powerful command, with which you can create much mayhem on your system. Like any powerful tool, it should be treated carefully and with respect. See the article on **fdisk** in the Lexicon for details on how to use this command.

Partitioning your hard drive can be an uncomplicated procedure. We offer these guidelines in an effort to make it as simple as possible. Before attempting any partitioning you should first back-up all the data currently on your hard drive. If you do not do this you risk losing data permanently. You should also know the correct physical parameters of your hard drive. This information can be obtained from your machine documentation or from the drive manufacturer. It is best not to rely on the parameters given in the BIOS: these may be translation parameters.

If your drive is formatted for MS-DOS, it is advisable to run MS-DOS **fdisk** before you start to install COHERENT. If the whole drive is taken up by DOS partitions, you must use MS-DOS **fdisk** to create a non-DOS area on the drive. It is not sufficient to have an empty MS-DOS logical drive set aside for COHERENT. COHERENT does not recognise MS-DOS logical drives, it only sees the whole partition. The following diagram shows the way the MS-DOS **fdisk** sees your drive:

```
============================
|    DOS Root Partition    |
|==========================|
|  DOS Extended Partition  |
|==========================|
|  =====================   |
|  |  Logical Drive 1  |   |
|  =====================   |
| - - - - - - - - - - - - - |
|  =====================   |
|  |  Logical Drive 2  |   |
|  =====================   |
| - - - - - - - - - - - - - |
|  =====================   |
|  |  Logical Drive 3  |   |
|  =====================   |
============================
```

And the following diagram shows the way the COHERENT **fdisk** sees your drive:

```
============================
|    DOS Root Partition    |
|==========================|
|                          |
|                          |
| DOS Extended Partition   |
|                          |
|                          |
============================
```

If you use COHERENT **fdisk** to repartition MS-DOS space, you risk causing MS-DOS **fdisk** to hang. One further word of warning. If you have an automated disk formatting and partitioning utility on your MS-DOS partition such as Disk Manager or Speedstor, you should operate it in "manual" mode, not in "automatic".

Some hard drives have more than 1,024 cylinders. COHERENT can only recognise a drive up to this limit. You may have a utility such as Speedstor that allows you to place MS-DOS partitions beyond that boundary. COHERENT will not see those partitions, but you can still access them as usual through MS-DOS.

When partitioning a drive with more than 1,024 cylinders, be sure to run the partitioning utility before you start to install COHERENT. You should create a non-DOS partition that falls completely within the 1,022-cylinder boundary. Your next MS-DOS partition should start no sooner than the 1,026th cylinder.

Adding a COHERENT Partition

The following describes how to add a new COHERENT partition on your hard disk.

During your initial installation of COHERENT, the installation program handled the details of preparing your hard disk for COHERENT. Adding a partition after the system is installed is not difficult, but it requires that you understand the operation of the following commands: **badscan**, **chmod**, **chown**, **fdisk**, **fsck**, **mkfs** and **mount**. See the Lexicon articles for each of these commands for further information before attempting to add a partition.

In general, the following steps are required when creating a partition for use by COHERENT. Please note that you must not change the size of your existing root partition, or you may no longer be able to boot COHERENT from the hard disk.

1. Completely back up all partitions on your hard disk. Be sure to back up the COHERENT partitions, as well as any non-COHERENT partitions (e.g., those for MS-DOS or OS/2). Verify that your backups are *readable* and *correct*.

2. Log in as the superuser **root**. Make sure all other users are off the system; then invoke the command **/etc/shutdown**. This shuts down COHERENT and returns the system to single-user mode. Type the command **sync** to flush all buffers.

3. Invoke the COHERENT command **fdisk** and add the COHERENT partition to your disk, as described above. Be sure to write down the device name associated with your new partition (e.g., **/dev/at0c**) and its size.

4. The command **badscan** checks the device for bad blocks. If your partition resides on a non-SCSI device (e.g., MFM, RLL, ESDI, or IDE), run the command **badscan** as follows:

```
/etc/badscan -v -o /conf/proto.device raw_device xdevice
```

where *device* specifies the four-character block-special device name for the partition (e.g., **at0c**), *raw_device* is the full device path name for the character-special device associated with the partition (e.g., **/dev/rat0c**), and *xdevice* specifies the partition-table device for the disk drive (e.g., **/dev/at0x**).

5. Invoke the command **mkfs** to create a COHERENT file system on the new partition, as follows:

   /etc/mkfs /dev/*device* /conf/proto.*device*

   This invocation will cause **mkfs** to use the contents of the "proto" file that **badscan** created when it built the *bad_block* list for the new partition.

6. If need be, use command **mkdir** to create a directory to use as a *mount point* for the newly created file system. The mount point is the directory onto which this directory's file system will be appended. Usually, this directory is located under '/', also called the *root directory*. You can, however, mount a file system onto any directory that already exists. If you create a new directory (e.g., **/w** or **/mydir**), use the commands **chown** and **chmod** to set an appropriate ownership and mode for for the directory.

7. Edit the file **/etc/mount.all** and add a line of the following form:

   /etc/mount *device* /*mount_point*

   where *device* is the full path name of the device that specifies your new partition (e.g., **/dev/at0c**), and *mount_point* is the name of the directory that you created in the earlier step.

8. Finally, edit the file **/etc/checklist** and add the character special device name (e.g., **/dev/rat0c**) of the new COHERENT partition to it. This will ensure that COHERENT will automatically run **fsck** on that partition's file system whenever you boot the system. This can be vital in recovering from a system crash.

Adding Another Hard Disk

If you wish to add another hard disk to your system, you may have to run some low-level routines that are hardware specific. See the documentation that accompanies your hardware for details.

In brief, when you install the hard disk, you must partition it, as you did your original hard disk when you first installed COHERENT. If you wish to add non-COHERENT operating systems to one or more partitions, do so first; then add COHERENT to the remaining partitions, as described above.

Changing the Size of the Root Partition

Changing the size of your **root** file system requires that you reinstall COHERENT. It is strongly advised that you back up *all* partitions of your system before you attempt to do this. In addition, to reduce the time involved in restoring your data files, make an additional backup of all directories and files that have changed form your original MWC installation. The command **find** will help you locate all such files; see its Lexicon entry for details.

You should then follow the directions given in the release notes for installing COHERENT. Note that when you attempt to install COHERENT over an existing COHERENT partition, COHERENT will ask you if you are sure you know what you're doing before the installation procedure creates a new file system on the partition. Be sure to request that a new file system be created, or the installation will fail.

After installing the COHERENT distribution onto your new root partition, restore any data files and directories from the second set of backups that you performed.

See Also

at, badscan, chmod, chown, fdisk, fsck, technical information, mkfs, mount, scsi

hash — Command

Add a command to the shell's hash table
hash [-r] [*command* ...]

The command **hash** lets you manipulate the Korn shell's hashing facility. A hashed command can be accessed instantly by the shell, without the delay of searching the directories in the **PATH** environmental variable.

When called with an argument, **hash** prints all hashed commands. When called with one or more *command* arguments, it adds *command* to its hash table. The option **-r** removes *command* from the hash table.

Note that before you can use hashing, you must use the **set** command to turn it on. For more information on the Korn shell's hashing feature, see the Lexicon entry for **ksh**.

See Also

commands, ksh

hdioctl.h — Header File

Control hard-disk I/O
#include <sys/hdioctl.h>

hdioctl.h declares constants and structures used to control hard-disk I/O.

See Also

header files

head — Command

Print the beginning of a file
head [+*n*[bcl]] [*file*]
head [-*n*[bcl]] [*file*]

head copies the first part of *file*, or of the standard input if none is named, to the standard output.

The given *number* tells **head** where to begin to copy the data. Numbers of the form *+number* measure the starting point from the beginning of the file; those of the form *-number* measure from the end of the file.

A specifier of blocks, characters, or lines (*b*, **c**, or **l**, respectively) may follow the number; the default is lines. If no *number* is specified, a default of +4 is assumed.

See Also

commands, dd, egrep, sed, tail

Notes

Because **head** buffers data measured from the end of the file, large counts may not work.

header files — Overview

A *header file* is a file of C code that contains definitions, declarations, and structures commonly used in a given situation. By tradition, a header file always has the suffix ".h". Header files are invoked within a C program by the command **#include**, which is read by **cpp**, the C preprocessor; for this reason, they are also called "include files".

Header files are one of the most useful tools available to a C programmer. They allow you to put into one place all of the information that the different modules of your program share. Proper use of header files will make your programs easier to maintain and to port to other environments.

COHERENT includes the following header files:

access.h . . . . . Check accessibility
acct.h. . . . . Format for process-accounting file
action.h . . . . . Describe parsing action and goto tables
sys/alloc.h . . . . . Define the allocator
ar.h . . . . . Format for archive files
ascii.h . . . . . Define non-printable ASCII characters
assert.h . . . . . Define **assert()**
sys/buf.h . . . . . Buffer header
canon.h . . . . . Portable layout of binary data
sys/chars.h . . . . . Character definitions
sys/con.h . . . . . Configure device drivers
sys/const.h . . . . . Declare machine-dependent constants
ctype.h. . . . . Header file for data tests
curses.h . . . . . Declare/define **curses** routines
sys/deftty.h. . . . . Default tty settings
sys/dir.h. . . . . Directory format
dirent.h . . . . . Define constant **dirent**
dumptape.h . . . . . Define data structures for dump tapes
ebcdic.h . . . . . Define constants for non-printable EBCDIC characters
errno.h. . . . . Error numbers used by errno()
sys/fblk.h . . . . . Define disk-free block
sys/fcntl.h . . . . . Manifest constants for file-handling functions"
sys/fd.h . . . . . Declare file-descriptor structure
sys/fdioctl.h . . . . . Control floppy-disk I/O
sys/fdisk.h . . . . . Fixed-disk constants and structures
sys/filsys.h . . . . . Structures and constants for super block
fperr.h . . . . . Constants used with floating-point exception codes
grp.h . . . . . Declare group structure
sys/hdioctl.h . . . . . Control hard-disk I/O
sys/ino.h . . . . . Constants and structures for i-nodes
sys/inode.h . . . . . Constants and structures for memory-resident i-nodes
sys/io.h . . . . . Constants and structures used by I/O
sys/ipc.h. . . . . Declarations for process communications
l.out.h . . . . . Object file format
limits.h . . . . . Define numerical limits
sys/lpioctl.h . . . . . Definitions for line-printer I/O control
sys/machine.h . . . . . Machine-dependent definitions
sys/malloc.h . . . . . Definitions for memory-allocation functions
math.h . . . . . Declare mathematics functions
sys/mdata.h . . . . . Define machine-specific magic numbers
mnttab.h. . . . . Structure for mount table
mon.h . . . . . Read profile output files
sys/mount.h . . . . . Define the mount table
mprec.h . . . . . Multiple-precision arithmetic
sys/msg.h . . . . . Definitions for message facility
sys/msig.h . . . . . Machine-dependent signals
mtab.h . . . . . Currently mounted file systems
sys/mtioctl.h. . . . . Magnetic-tape I/O control

mtype.h . . . . . List processor code numbers
n.out.h . . . . . Define n.out file structure
sys/param.h . . . . . Define machine-specific parameters
path.h . . . . . Define/declare constants and functions used with path
sys/poll.h . . . . . Define structures/constants used with polling devices
sys/proc.h. . . . . Define structures/constants used with processes
pwd.h. . . . . Declare password structure
sys/sched.h. . . . . Define constants used with scheduling
sys/seg.h . . . . . Definitions used with segmentation
sys/sem.h. . . . . Definitions used by semaphore facility
setjmp.h . . . . . Define setjmp() and longjmp()
sgtty.h . . . . . Definitions used to control terminal I/O
sys/shm.h. . . . . Definitions used with shared memory
signal.h . . . . . Declare signals
sys/stat.h . . . . . Definitions and declarations used to obtain file status
stdarg.h . . . . . Declare/define routines for variable arguments
stddef.h . . . . . Declare/define standard definitions
stdio.h . . . . . Declarations and definitions for I/O
stdlib.h. . . . . Declare/define general functions
sys/stream.h . . . . . Definitions for message facility"
string.h . . . . . Declare string functions
termio.h . . . . . Definitions used with terminal input and output
time.h . . . . . Give time-description structure
sys/timeb.h. . . . . Declare timeb structure
timef.h . . . . . Definitions for user-level timed functions
sys/timeout.h . . . . . Define the timer queue
sys/times.h . . . . . Definitions used with times() system call
sys/tty.h. . . . . Define flags used with tty processing
sys/types.h . . . . . Declare system-specific data types
sys/uproc.h . . . . . Definitions used with user processes
utmp.h . . . . . Login accounting information
sys/utsname.h . . . . . Define utsname structure
v7sgtty.h . . . . . UNIX Version 7-style terminal I/O

See Also

C language, #include, portability

help — Command

Print concise description of command
help *command*

help prints a concise description of the options available for each specifed *command*. If the *command* is omitted, **help** prints a simple description of itself, followed by information about the command given by **$LASTERROR**, which is the last command returning a nonzero exit status.

help provides more information than the usage message printed by a command, but less than the detailed description given by the **man** command. The primary purpose of **help** is to refresh your memory if you have forgotten an option to *command*.

help looks in **/etc/helpfile** for system information and the file named in environmental variable **$HELP** for user-specific information. Information about a *command* begins with a line

#command

and ends with the next line beginning with '#' in **/etc/helpfile** or **$HELP. help** constructs the index file **/etc/helpindex** to make subsequent searches of **/etc/helpfile** faster.

Files

/etc/helpfile — Additional system information
/etc/helpindex — Index for helpfile
$HELP — User information
$LASTERROR — Default command help

See Also

commands, man

HOME — Environmental Variable

User's home directory
HOME=*home directory*

The environmental variable **HOME** name's the user's home directory. Some commands use this name by default if they require the name of a directory and none is supplied. For example, if you type the change directory command **cd** without an argument, it will change the current directory to the one named by the **HOME**.

See Also

environmental variables

hp — Command

Prepare files for Hewlett-Packard LaserJet printer
hp [**-acflr**] [**-i***marg*] [**-t***top*] [**-p***lines*] [*file* ...]

The command **hp** translates **nroff** font specifications into the correct escape sequences for an HP LaserJet compatible printer. It also allows the user to set indentation, page length, landscape mode, and so on. Because some LaserJet printers stack pages in reverse order as they are printed, **hp** can put pages out in reverse order.

hp recognizes the following options:

-f Print pages in the normal order. This is the default.

-i*marg* Set the page indentation to *marg*.

-l Print pages in landscape mode.

-p*lines* Set the page length to *lines*.

-r Print pages in reverse order (for LaserJet I).

-t*top* Set the top margin to *top*.

Example

To generate listings of all C programs in the current directory, enter the command

```
pr *.c | hp | hpr -B
```

See Also

commands, hpd, printer

hpd — System Maintenance

Hewlett-Packard LaserJet printer spooler daemon
/usr/lib/hpd

hpd is a daemon program that runs in the background and prints listings queued by the **hpr** command. **hpd** is run automatically by **hpr**. If there is no printing to do, or if another daemon is already running (indicated by the **dpid** file), **hpd** exits immediately. Otherwise, it searches the spool directory for control files of listings to print. These control files contain the names of files to print, the user name, banner pages, and files to be removed upon completion.

hpd does not print listings in any particular order. There is no prioritization of printing, either by size or by requester.

Files

/dev/rhp — Raw device for LaserJet printer
/usr/spool/hpd — Spool directory
/usr/spool/hpd/cf* — Control files
/usr/spool/hpd/df* — Data files
/usr/spool/hpd/dpid — Lock and process id

See Also

hpr, hpskip, init, lpd, printer, system maintenance

hpr — Command

Send file to Hewlett-Packard LaserJet printer spooler
hpr [-Bcemnr] [-b *banner*] [**-f** *fontnum*] [*file* ...]

hpr lets you print each specified *file* on the Hewlett-Packard LaserJet printer, without conflicting with printing by other users. If no *file* is specified, **hpr** prints the standard input on the LaserJet printer.

hpr recognizes the following options:

-B Suppress printing of a banner page.

-b The next argument is the banner.

-c Copy the files (allowing changes to be made before the printing completes).

-e Erase all "soft fonts" from the printer's memory.

-f *fontnum file1 ... fileN*
Load the Hewlett-Packard "soft fonts" stored in files *file1* through *fileN* into the printer's memory; set the font identifiers to begin at *fontnum*.

-m Send a message when the printing completes.

-n Do not send a message (default).

-r Remove the files when they have been spooled.

hpskip terminates or restarts the current listing. **hp** converts **nroff** output into forms usable by the LaserJet; it is also used to describe the format of the printing.

Examples

To print the file **foo** on the LaserJet, type:

```
hpr foo
```

The following example loads the soft fonts in files **foo**, **bar**, and **baz** into the printer's memory, and sets their font identifiers to begin at 15:

```
hpr -f 15 foo bar baz
```

Files

/dev/rhp — Raw device for LaserJet printer
/usr/lib/hpd — Line-printer daemon for LaserJet printer
/usr/spool/hpd — Spool directory for LaserJet printer
/usr/spool/hpd/dpid — Daemon lockfile

See Also

commands, hp, hpd, hpskip, lpr, pr, printer

hpskip — Command

Abort/restart current listing on Hewlett-Packard LaserJet
hpskip [-r]

hpskip gives some control over printing with the LaserJet printer spooler. When invoked without the **-r** option, **hpskip** terminates the current listing with a message. When invoked with the **-r** option, **hpskip** restarts the current listing.

hpr spools files to the LaserJet printer.

Files

/usr/lib/hpd — LaserJet printer daemon
/usr/spool/hpd — Spool directory
/usr/spool/hpd/dpid — Daemon lockfile

See Also

commands, hpd, hpr, lpskip, pr

hs — Device Driver

Device driver for polled serial ports

The COHERENT **hs** driver adds support for up to eight serial lines, **/dev/hs00** through **/dev/hs07**.

Serial lines controlled via the **hs** driver can be opened in one of two ways, as follows:

/dev/hs??
: Polled, local mode (no modem control).

/dev/hs??**r**
: Polled, remote mode (modem control).

Any port used with the **hs** device driver will be polled, i.e., interrupt operation is not used. Please refer to the Lexicon article **com** for explanations of "local" vs "remote" and "polled" vs "interrupt-driven".

To use the **hs** driver, first configure it to match your equipment (see below), then load the driver using the following command while running as the superuser **root**:

```
/etc/drvld -r /drv/hs
```

To unload the driver without rebooting COHERENT, first use the **ps** command with the **-d** option to get the process identifier for the **hs** driver process, then unload the driver process by using the **kill** command. Note that the **hs** driver process will not unload until all **open**ed ports have been closed. For example (user input shown in bold):

```
$ ps -d
TTY        PID
-------      0  <idle>
-------     38  <hs>
...
$ kill kill 38
```

The present version of COHERENT limits "polled" operation to one device driver at a time. Therefore, if any of the **com** family of devices is used in polled mode, **hs** devices cannot be used. Conversely, **/dev/com1pl** through **/dev/com4pl** and **/dev/com1pr** through **/dev/com4pr** cannot be used if the **hs** driver is in use. Both drivers can be present at the same time, but polled devices may not be **open** under both drivers at the same time. Note that enabling a port via **/etc/enable** keeps it open continuously.

Port Configuration

The default configuration for the **hs** driver is for four ports, at hexadecimal addresses 0x3F8, 0x2F8, 0x3E8, and 0x2E8, at a speed of 9600 baud. The driver is configured by setting the following parameters:

1. The number of ports.
2. The I/O address for each port.
3. The default speed of each port.

All steps in the configuration must be done as the superuser **root**. Patch the number of ports into driver variable **HSNUM_**. For example, if you wish to support three ports, enter:

```
/conf/patch /drv/hs HSNUM_=3
```

Address and speed information are stored sequentially starting at variable **HS_PORTS_**. The speed for each port is indicated by the corresponding value found in **<sgtty.h>**, from one, corresponding to 50 baud, to 16, corresponding to 9600 baud. If the three ports in the example above are at hexadecimal adresses of 0x2A0, 0x2B0, and 0x2C0, with speeds of 2400, 2400, and 9600 baud, respectively, then the following three patches must be performed:

```
/conf/patch /drv/hs HS_PORTS_=0x2A0 HS_PORTS_+2=12
/conf/patch /drv/hs HS_PORTS_+4=0x2B0 HS_PORTS_+6=12
/conf/patch /drv/hs HS_PORTS_+8=0x2C0 HS_PORTS_+10=16
```

Finally, nodes must be created for each port using the **mknod** command. The major device number is 7; the minor number will range from 0 through 7 for ports **/dev/hs00** through **/dev/hs07**, respectively, with 128 added to the device minor number if modem control is desired. The following commands will make nodes in **/dev** for local and remote versions of the three ports in the example:

```
/etc/mknod -f /dev/hs00  c  7  0
/etc/mknod -f /dev/hs01  c  7  1
/etc/mknod -f /dev/hs02  c  7  2
/etc/mknod -f /dev/hs00r c  7  128
/etc/mknod -f /dev/hs01r c  7  129
/etc/mknod -f /dev/hs02r c  7  130
```

See Also

com, device drivers, drvld

Diagnostics

An attempt to open a non-existent device will generate error messages. This can occur if hardware is absent or not turned on.

Notes

Note that if any **com** device driver is used in polling mode, the **hs** driver cannot be used, and vice versa.

hypot() — Mathematics Function (libm)

Compute hypotenuse of right triangle
#include <math.h>
double hypot(*x*, *y*) **double** *x*, *y*;

hypot computes the hypotenuse, or distance from the origin, of its arguments *x* and *y*. The result is the square root of the sum of the squares of *x* and *y*.

Example

For an example of this function, see the entry for **acos**.

See Also

cabs(), mathematics library

I

i-node — Definition

COHERENT system file identifier

Each file on a COHERENT file system is identified by a unique number, called an *i-node number* or *i-number*. Each i-node contains information about a file: its mode, link count, user identifier, group identifier, size, location on the file system, access time, modify time, and creation time.

The user refers to a file by a file name, stored in a directory; the directory entry identifies the file by its i-node number. A device and i-node number together uniquely specify a file. The headers **ino.h** and **i-node.h** define, respectively, disk i-nodes and memory i-nodes.

See Also

definitions

icheck — Command

i-node consistency check
icheck [-s] [-b *N* ...] [-v]*filesystem* ...

Each block in a file system must be either free (i.e., in the free list) or allocated (i.e., associated with exactly one i-node). **icheck** examines each specified *filesystem*, printing block numbers that are claimed by more than one i-node, or claimed by both an i-node and the free list. It also checks for blocks that appear more than once in the block list of an i-node or in the free list.

The option **-v** (verbose) causes **icheck** to print a summary of block usage in the *filesystem*. The option **-s** causes **icheck** to ignore the free list, to note which blocks are claimed by i-nodes, and to rebuild the free list with the remainder. A list of block numbers may be submitted with the **-b** flag; **icheck** prints the data structure associated with each block as the file system is scanned.

The raw device should be used, and the *filesystem* should be unmounted if possible. If this is not possible (e.g., on the root file system) and the **-s** option is used, the system must be rebooted immediately to expunge the obsolete superblock.

The exit status bits for a bad return are as follows:

0x01 Miscellaneous error (e.g. out of space)
0x02 Too hard to fix without human intervention
0x04 Bad free block
0x08 Missing blocks
0x10 Duplicates in free list
0x20 Bad block in free list

See Also

clri, commands, dcheck, fsck, ncheck, sync, umount

Diagnostics

The message "dups in free" indicates a block is in the free list more than once. "bad freelist" indicates the presence of bad blocks on the free list. A "bad" block is one that lies outside the bounds of the file system. A "dup" (duplicated) block is one associated with the free list and an i-node, or with more than one i-node. All the errors above *must* be corrected before the file system is mounted. "bad ifree" means allocated i-nodes are on the free i-node list; this is inconsequential.

This command has largely been replaced by **fsck**.

if — C Keyword

Introduce a conditional statement

if is a C keyword that introduces a conditional statement. For example,

```
if (i==10)
     dosomething();
```

will **dosomething** only if **i** equals ten.

if statements can be used with the statements **else if** and **else** to create a chain of conditional statements. Such a chain can include any number of **else if** statements, but only one **else** statement.

See Also

C keywords, else

if — Command

Execute a command conditionally

if *sequence1* **then** *sequence2* [**elif** *sequence3* **then** *sequence4*] ... [**else** *sequence5*] **fi**

The shell construct **if** executes commands conditionally, depending on the exit status of the execution of other commands.

First, **if** executes the commands in *sequence1*. If the exit status is zero, it executes the commands in *sequence2* and terminates. Otherwise, it executes the optional *sequence3* if given, and executes *sequence4* if the exit status is zero. It executes additional **elif** clauses similarly. If the exit status of each tested command sequence is nonzero, it executes the optional **else** part *sequence5*.

Because the shell recognizes a reserved word only as the unquoted first word of a command, each **then**, **elif**, **else**, and **fi** must either occur unquoted at the start of a line or be preceded by ';'.

The shell executes **if** directly.

Example

For an example of this command, see the entry for **trap**.

See Also

commands, ksh, sh, test

IFS — Environmental Variable

Characters recognized as white space

The environmental variable **IFS** lists the characters that the shell recognizes as white space.

See Also

environmental variables, ksh, sh

index() — String Function (libc)

Find a character in a string
char *index(*string*, *c***) char ***string*; **char** *c*;

index scans the given *string* for the first occurrence of the character *c*. If *c* is found, **index** returns a pointer to it. If it is not found, **index** returns NULL.

Note that having **index** search for a null character will always produce a pointer to the end of a string. For example,

```
char *string;
assert(index(string, 0)==string+strlen(string));
```

will never fail.

Example

For an example of this function, see the entry for **strncpy**.

See Also

pnmatch(), rindex(), string functions

Notes

This function is identical to the function **strchr**, which is described in the ANSI standard. COHERENT includes **strchr** in its libraries. It is recommended that it be used instead of **index** so that programs more closely approach strict conformity with the ANSI standard.

init — System Maintenance

System initialization
/etc/init

The COHERENT boot procedure executes **init** as process 1 to perform initialization. **init** opens the console terminal **/dev/console** and invokes the shell script **/etc/brc** if it exists. If it does not, **init** invokes a shell **sh** on it with **HOME** set to **/etc**. The shell executes **/etc/profile** and **/etc/.profile** if present. The system then runs in single-user mode and accepts commands from the console.

When the console terminates the shell, normally by typing **<ctrl-D>**, **init** brings up the system in multiuser mode. It executes the shell command file **/etc/rc**, which performs standard bookkeeping and maintenance chores. Typically it mounts standard file systems, removes temporary files, and invokes **cron** and **update**. If desired, it may load device drivers, enable swapping with **swap**, and enable process accounting with **accton**.

Next, **init** opens terminals as specified in the file **/etc/ttys**. It invokes **getty** to read a user name and perform a **login** on each terminal.

When a user shell terminates, **init** updates the system accounting information in **/etc/utmp** and **/usr/adm/wtmp**. Then it reopens the appropriate terminal and invokes **getty**, as above.

init rescans the file **/etc/ttys** for terminal changes if it receives the signal **SIGQUIT**. The command **kill quit 1** sends **SIGQUIT** to the **init** process. **init** then invokes **getty** as necessary.

init returns the system to single user mode if it receives the signal **SIGHUP**. The command **kill -1 1** sends **SIGHUP** to the **init** process.

Files

/dev/console — Console terminal
/dev/tty?? — Terminal devices
/etc/brc — Boot command file
/etc/rc — initialization command file
/etc/ttys — Active terminals
/etc/utmp — Logged in users
/usr/adm/wtmp — Login accounting data
/usr/spool/uucp/LCK..* — Terminal locks

See Also

getty, kill, login, sh, system maintenance, ttys

initialization — Definition

The term *initialization* refers to setting a variable to its first, or initial, value.

Rules of Initialization

Initializers follow the same rules for type and conversion as do assignment statements.

If a static object with a scalar type is not explicitly initialized, it is initialized to zero by default. Likewise, if a static pointer is not explicitly initialized, it is initialized to NULL by default. If an object with automatic storage duration is not explicitly initialized, its contents are indeterminate.

Initializers on static objects must be constant expressions; greater flexibility is allowed for initializers of automatic variables. These latter initializers can be arbitrary expressions, not just constant expressions. For example,

```
double dsin = sin(30.0);
```

is a valid initializer, where **dsin** is declared inside a function.

To initialize an object, use the assignment operator '='. The following sections describe how to initialize different classes of objects.

Scalars

To initialize a scalar object, assign it the value of a expression. The expression may be enclosed within braces; doing so does not affect the value of the assignment. For example, the expressions

```
int example = 7+12;
```

and

```
int example = { 7+12 };
```

are equivalent.

Unions and Structures

The initialization of a **union** by definition fills only its *first* member.

To initialize a **union**, use an expression that is enclosed within braces:

```
union example_u {
     int member1;
     long member2;
     float member3;
} = { 5 };
```

This initializes **member1** to five. That is to say, the **union** is filled with an **int**-sized object whose value is five.

To initialize a structure, use a list of constants or expressions that are enclosed within braces. For example:

```
struct example_s {
     int member1;
     long member2;
     union example_u member3;
};

struct example_s test1 = { 5, 3, 15 };
```

This initializes **member1** to five, initializes **member2** to three, and initializes the *first* member of **member3** to 15.

Strings and Wide Characters

To initialize a string pointer or an array of wide characters, use a string literal.

The following initializes a string:

```
char string[] = "This is a string";
```

The length of the character array is 17 characters: one for every character in the given string literal plus one for the null character that marks the end of the string.

If you wish, you can fix the length of a character array. In this case, the null character is appended to the end of the string only if there is room in the array. For example, the following

```
char string[16] = "This is a string";
```

writes the text into the array **string**, but does not include the concluding null character because there is not enough room for it.

A pointer to **char** can also be initialized when the pointer is declared. For example:

```
char *strptr = "This is a string";
```

initializes **strptr** to point to the first character in **This is a string**. This declaration automatically allocates exactly enough storage to hold the given string literal, plus the terminating null character.

Arrays

To initialize an array, use a list of expressions that is enclosed within braces. For example, the expression

```
int array[] = { 1, 2, 3 };
```

initializes **array**. Because **array** does not have a declared number of elements, the initialization fixes its number of elements at three. The elements of the array are initialized in the order in which the elements of the initialization list appear. For example, **array[0]** is initialized to one, **array[1]** to two, and **array[2]** to three.

If an array has a fixed length and the initialization list does not contain enough initializers to initialize every element, then the remaining elements are initialized in the default manner: static variables are initialized to zero, and other variables to whatever happens to be in memory. For example, the following:

```
int array[3] = { 1, 2 };
```

initializes **array[0]** to one, **array[1]** to two, and **array[2]** to zero.

The initialization of a multi-dimensional array is something of a science in itself. The ANSI Standard defines that the ranks in an array are filled from right to left. For example, consider the array:

```
int example[2][3][4];
```

This array contains two groups of three elements, each of which consists of four elements. Initialization of this array will proceed from **example[0][0][0]** through **example[0][0][3]**; then from **example[0][1][0]** through **example[0][1][3]**; and so on, until the array is filled.

It is easy to check initialization when there is one initializer for each "slot" in the array; e.g.,

```
int example[2][3] = {
      1,  2,  3,  4,  5,  6
};
```

or:

```
int example[2][3] = {
     {  1,  2,  3 }, {  4,  5,  6 }
};
```

The situation becomes more difficult when an array is only partially initialized; e.g.,

```
int example[2][3] = {
        { 1 }, { 2, 3 }
};
```

which is equivalent to:

```
int example[2][3] = {
     { 1, 0, 0 }, { 2, 3, 0 }
};
```

As can be seen, braces mark the end of initialization for a "cluster" of elements within an array. For example, the following:

```
int example[2][3][4] = {
     5, { 1, 2 }, { 5, 2, 4, 3 }, { 9, 9, 5 },
     { 2, 3, 7 } };
```

is equivalent to entering:

```
int example[2][3][4] = {
     { 5, 0, 0, 0 },
     { 1, 2, 0, 0 },
     { 5, 2, 4, 3 },

     { 9, 9, 5, 0 },
     { 2, 3, 7, 0 },
     { 0, 0, 0, 0 }
};
```

The braces end the initialization of one cluster of elements; the next cluster is then initialized. Any elements within a cluster that have not yet been initialized when the brace is read are initialized in the default manner.

See Also

array, C language, definitions, struct, union

ino.h — Header File

Constants and structures for disk i-nodes
#include <sys/inode.h>

inode.h declares structures and constants that are used to describe i-nodes.

See Also

i-node, header files

inode.h — Header File

Constants and structures for memory-resident i-nodes
#include <sys/inode.h>

inode.h declares structures and constants for memory-resident i-nodes.

See Also

header files, i-node

install — Command

Install a software update onto COHERENT
/etc/install *id device ndisks*

The command **install** installs an update of the COHERENT system onto your hard disk. *id* identifies the update to be installed. *device* is the device from which the update disks will be read. *ndisks* is the number of disks that comprise the update.

Third-Party Software

install also provides a standard mechanism by which software developers can install their software onto systems that run COHERENT. The rest of this article discusses how to prepare a software release so that it can be installed using **install**.

For **install** to be able to install a software distribution, the distribution must consist of a set of mountable floppy disks, each holding a COHERENT file system created by **mkfs**. This keeps the disks independent of each other and also lets the user to insert the disks in any order. **install** records the fact that it has read a given disk from the distribution, thus preventing the user from attempting to read a given disk more than once during an installation session.

Floppy disks should be built using **mkfs**, with possible input being generated by the command **unmkfs**. Each disk in the distribution must hold in its root directory a file whose name is of the form:

/*id.sequence*

Here, *id* identifies the release, as described above. Note that *id* must be formed from the set of upper- and lower-case letters, digits, the period '.', and the underscore character '_', and not exceed nine characters in length. *sequence* indicates which disk in the distribution this disk is, from one through the total number of disks.

install uses the command **cpdir** to copy each of the distribution disks to directory / on the

current system. Therefore, all disks should be "root based" (i.e., full path names should be used). Because **install** is run by the superuser, **cpdir** preserves the date and time for each file, and preserves ownership and modes. To keep file ownership consistent with COHERENT conventions, make files that are neither *setuid* nor *setgid* owned by user **bin** and group **bin**. Directories found on the distribution disks will be created on the target file system, as needed. Be careful when choosing the ownership and mode of directories because you could inadvertently compromise the security of your users' systems.

Postprocessing

After all disks in a distribution have been successfully copied by the user, **install** checks for the existence of a file of the form

```
/conf/id.post
```

where *id* matches the *id* field found on the **install** command line. If found, **install** executes this file to allow special "postprocessing," such as installing manual pages into directory **/usr/man** or executing installation-specific commands.

Before an installation procedure completes its postprocessing, it should remove any *id* files of the following form from the target system:

/conf/*id*.post
/*id.sequence*

Adding Manual-Page Entries

As part of building a distribution, you usually must generate pre-processed or "cooked" manual-page entries for distribution with your upgrade or add-on package. These should be inserted into the subdirectories of **/usr/man**, with the name of the subdirectory being specific to your product. This naming convention avoids name-space collisions, should multiple applications use the same name for a manual-page entry.

If you install new or additional manual pages, you must update the index file used by the **man** command to locate manual entries. File **/usr/man/man.index** on the target file system contains index entries for all manual pages on the system. As part of postprocessing, you must append index information for your manual pages to the end of the existing index file. In addition, file **/usr/man/man.help** contains the **man** command's help message. This includes a list of valid topics and some explanatory text. You should also append to this file a brief list of the manual page entries that you have added. For further information on manual pages, see the Lexicon entry for the command **man**.

Logging

install logs all partial as well as completed installations in file **/etc/install.log**. This information includes date/time stamps and the command-line arguments to **install**.

Example

The following installs COHERENT update **coh.301**, which consists of one disk, from a high-density 5.25-inch floppy drive:

```
/etc/install coh.301 /dev/fha0 1
```

Files

/etc/install.log

See Also

commands, man, mkfs, unmkfs

int — C Keyword

Data type

An **int** is the most commonly used numeric data type, and is normally used to encode integers. With COHERENT, **sizeof int** equals 2, that is, two **chars** (15 bits plus a sign bit); therefore, an **int** can contain values from -32768 to +32767. An **int** normally is sign extended when cast to a larger data type; an **unsigned int**, however, will be zero extended.

See Also

C keywords, data formats, data types, long

interrupt — Definition

An **interrupt** is an interruption of the sequential flow of a program. It can be generated by the hardware, from within the program itself, or from the operating system.

See Also

definitions, signal()

io.h — Header File

Constants and structures used by I/O
#include <sys/io.h>

io.h declares constants and structures used by various I/O routines.

See Also

header files

ioctl() — COHERENT System Call

Device-dependent control
ioctl(*fd, command, info***)**
int *fd, command*;
char **info*;

ioctl provides direct interaction with a device driver. Possible uses include setting or retrieving parameters for devices (line printers, communications lines, terminals) and non-standard spacing operations for tape drives.

ioctl acts upon a block special file or a character special file associated with the already open file descriptor *fd*. *command* points to the specific request. A system header file defines symbolic command parameters for each device type. For example, **sgtty.h** defines commands for terminals and **mtioctl.h** defines commands for magnetic tape drives. Using the symbolic command definitions from the header files promotes device independence within each device type. The entry for **device drivers** names other sections that detail specific devices.

info passes a buffer of information (defined by structures in the appropriate header files) to the driver. For any *command* not needing additional information, this argument should be NULL.

Some **ioctl** requests work on all files, and are not passed to any driver.

Files

<sgtty.h>
<lpioctl.h>
<mtioctl.h>

See Also

COHERENT system calls, exec, getty, open(), read(), sgtty, stty, write()

Diagnostics

ioctl returns -1 on errors, such as a bad file descriptor. Because the call is device dependent, almost any other error could be returned.

Notes

The type of the *info* argument to **ioctl** is declared as **char *** mainly for portability reasons. In most cases, the actual argument type will be something like **struct sgttyb ***, depending on the particular device and command. The actual argument should be cast to type **char *** to ensure cross-machine portability.

ipc.h — Header File

Definitions for process communications
#include <sys/ipc.h>

ipc.h defines constants and structures used by functions that perform inter-process communications.

See Also

header files

isalnum() — ctype Macro (ctype.h)

Check if a character is a number or letter
#include <ctype.h>
int isalnum(*c***) int** *c***;**

isalnum tests whether the argument *c* is alphanumeric (**0-9**, **A-Z**, or **a-z**). It returns a number other than zero if *c* is of the desired type, and zero if it is not. **isalnum** assumes that *c* is an ASCII character or EOF.

Example

For an example of how to use this macro, see the entry for **ctype**.

See Also

ASCII, ctype

isalpha() — ctype Macro (ctype.h)

Check if a character is a letter
#include <ctype.h>
int isalpha(*c***) int** *c***;**

isalpha tests whether the argument *c* is a letter (**A-Z** or **a-z**). It returns a number other than zero if *c* is an alphabetic character, and zero if it is not. **isalpha** assumes that *c* is an ASCII character or EOF.

Example

For an example of this macro, see the entry for **ctype**.

See Also

ASCII, ctype

isascii() — ctype Macro (ctype.h)

Check if a character is an ASCII character
#include <ctype.h>
int isascii(*c*) int *c*;

isascii tests whether the argument *c* is an ASCII character (0 <= *c* <= 0177). It returns a number other than zero if *c* is an ASCII character, and zero if it is not. Many other **ctype** macros will fail if passed a non-ASCII value other than EOF.

Example

For an example of how to use this macro, see the entry for **ctype**.

See Also

ASCII, ctype

isatty() — General Function (libc)

Check if a device is a terminal
int isatty(*fd*) int *fd*;

isatty checks to see if a device is a terminal. It returns one if the file descriptor *fd* describes a terminal, and zero otherwise.

Files

/dev/* — Terminal special files
/etc/ttys — Login terminals

See Also

general functions, ioctl(), tty, ttyname(), ttyslot()

iscntrl() — ctype Macro (ctype.h)

Check if a character is a control character
#include <ctype.h>
int iscntrl(*c*) int *c*;

iscntrl tests whether the argument *c* is a control character (including a newline character) or a delete character. It returns a number other than zero if *c* is a control character, and zero if it is not. **iscntrl** assumes that *c* is an ASCII character or **EOF**.

Example

For an example of how to use this macro, see the entry for **ctype**.

See Also

ctype

isdigit() — ctype Macro (ctype.h)

Check if a character is a numeral
#include <ctype.h>
int isdigit(*c*) int *c*;

isdigit tests whether the argument *c* is a numeral (**0-9**). It returns a number other than zero if *c* is a numeral, and zero if it is not. **isdigit** assumes that *c* is an ASCII character or **EOF**.

Example

For an example of how to use this macro, see the entry for **ctype**.

See Also

ASCII, ctype

islower() — ctype Macro (ctype.h)

Check if a character is a lower-case letter

#include <ctype.h>
int islower(*c*) int *c*;

islower tests whether the argument *c* is a lower-case letter (**a-z**). It returns a number other than zero if *c* is is a lower-case letter, and zero if it is not. **islower** assumes that *c* is an ASCII character or **EOF**.

Example

For an example of how to use this macro, see the entry for **ctype**.

See Also

ASCII, ctype

ispos() — Multiple-Precision Mathematics

Return if variable is positive or negative

#include <mprec.h>
int ispos(*a*)
mint **a*;

The COHERENT system includes a suite of routines that allow you to perform multiple-precision mathematics. **ispos** returns true (nonzero) if *a* is not negative, false (zero) if *a* is negative.

See Also

multiple-precision mathematics

isprint() — ctype Macro (ctype.h)

Check if a character is printable

#include <ctype.h>
int isprint(*c*) int *c*;

isprint is a macro that tests if *c* is printable, i.e, if it is neither a delete nor a control character. It returns a number other than zero if *c* is a printable character, and zero if it is not. **isprint** assumes that *c* is an ASCII character or **EOF**.

Example

For an example of how to use this macro, see the entry for **ctype**.

See Also

ASCII, ctype

ispunct() — ctype Macro (ctype.h)

Check if a character is a punctuation mark

#include <ctype.h>
int ispunct(*c*) int *c*;

ispunct tests whether the argument *c* is a punctuation mark, i.e., neither an alphanumeric character nor a control character. It returns a number other than zero if the character

tested is a punctuation mark, and zero if it is not. **ispunct** assumes that *c* is an ASCII character or **EOF**.

Example

For an example of how to use this macro, see the entry for **ctype**.

See Also

ASCII, ctype

isspace() — ctype Macro (ctype.h)

Check if a character prints white space

#include <ctype.h>
int isspace(*c***) int** *c***;**

isspace tests whether the argument *c* is a space, tab, newline, carriage return, or form-feed character. It returns a number other than zero if *c* is a white-space character, and zero if it is not. **isspace** assumes that *c* is an ASCII character or EOF.

Example

For an example of how to use this macro, see the entry for **ctype**.

See Also

ASCII, ctype

isupper() — ctype Macro (ctype.h)

Check if a character is an upper-case letter

#include <ctype.h>
int isupper(*c***) int** *c***;**

isupper tests whether the argument *c* is an upper-case letter (**A-Z**). It returns a number other than zero if *c* is an upper-case letter, and zero if it is not. **isupper** assumes that *c* is an ASCII character or **EOF**.

Example

For an example of how to use this macro, see the entry for **ctype**.

See Also

ASCII, ctype

itom() — Multiple-Precision Mathematics

Create a multiple-precision integer

#include <mprec.h>
mint *itom(*n***)**
int *n***;**

The COHERENT system includes a suite of routines that allow you to perform multiple-precision mathematics. **itom** creates a new multiple-precision integer (or **mint**), initializes it to the signed integer value *n*, and returns a pointer to it. You can use the function **mintfr** to reclaim the storage used by the **mint** created by **itom**.

See Also

multiple-precision mathematics

J

j0() — Mathematics Function (libm)

Compute Bessel function
#include <math.h>
double j0(z) double z;

j0 computes the Bessel function of the first kind for order 0 for its argument z.

Example

This example, called **bessel.c**, demonstrates the Bessel functions **j0**, **j1**, and **jn**. Compile it with the following command line

```
cc -f bessel.c -lm
```

to include floating-point functions and the mathematics library.

```
#include <math.h>
#include <stdio.h>
#define display(x) dodisplay((double)(x), #x)

dodisplay(value, name)
double value; char *name;
{
     if (errno)
          perror(name);

     else
          printf("%10g %s\n", value, name);
     errno = 0;
}

main()
{
     extern char *gets();
     double x;
     char string[64];

     for(;;) {
          printf("Enter number: ");
          if(gets(string) == NULL)
               break;
          x = atof(string);

          display(x);
          display(j0(x));
          display(j1(x));
          display(jn(0,x));
```

```
            display(jn(1,x));
            display(jn(2,x));
            display(jn(3,x));
        }
        putchar('\n');
}
```

See Also

j1(), jn(), mathematics library

j1() — Mathematics Function (libm)

Compute Bessel function
#include <math.h>
double j1(*z***) double** *z***;**

j1 takes *z* and computes the Bessel function of the first kind for order 1.

Example

For an example of this function, see the entry for **j0**.

See Also

j0(), jn(), mathematics library

jn() — Mathematics Function (libm)

Compute Bessel function
#include <math.h>
double jn(*n*, *z***) int** *n***; double** *z***;**

jn takes *z* and computes the Bessel function of the first kind for order *n*.

Example

For an example of this function, see the entry for **j0**.

See Also

j0(), j1(), mathematics library

jobs — Command

Print information about jobs
jobs

The command **jobs** is used with the Korn shell's job-control feature. It prints information about all background jobs. The information printed is in the following format:

%num [+-] pid status command

num indicates the job number, + indicates that the job is the "current job"; - indicates that it is the "previous job". *pid* gives the process identifier of the job. *status* indicates the status of the job. *command* gives the stopped job's command line.

For details about job control, see the Lexicon entry for **ksh**.

See Also

commands, ksh

join — Command

Join two data bases

join [-**a** [*n*]] [-**e** *string*] [-**j**[*n*] *keyf*] [-**o** *n.m* ...] [-**t***c*]*file1 file2*

join processes the text files *file1* and *file2*, each of which contains a relational data base. If either file name is '-', the standard input is used for that file.

For the purposes of **join**, a data base file contains a set of records, one per input line. Each record contains a number of *fields*. One field is differentiated as *key* field for each file. Each file must be sorted by key field, for example with **sort**.

By default, the key field is the first field in each record. The **-j** option changes the key field number to *keyf* for the desired file. In this and other options below, the optional file number *n* must be **1** to indicate *file1* or **2** to indicate *file2*. If no *n* is given, both *file1* and *file2* are assumed.

Normally, fields are separated by any amount of white space (blanks or tabs). Leading blanks or tabs are not considered part of the fields. With the **-t** option, the separator character is *c*. With this option zero-length fields are possible; every occurrence of the separator ends the previous field and starts a new one.

Output consists only of records for which the key field occurs in both files. As a consequence of the sorted order of the input, the output is also sorted by the key field. Each output record has first the key field, then each field from the *file1* record but the key field, and then each field from the *file2* record but the key field. Fields are separated in the output with the specified field character, or with a space character if no **-t** option was given. Output records are always terminated with a newline. Under the **-e** option, *string* is printed for each empty field.

The **-a** option enables printing of records found in only file *n*. If *n* is missing, unpaired records are printed from both input files. To output only certain fields, the **-o** option precedes a list of desired fields to print. Each element is of the form *n.m* where *n* is the file number and *m* is the field number.

For example,

```
join -t: -j1 3 -o 1.3 2.4 1.4 1.1 2.2 filea fileb
```

joins **filea** and **fileb** which have fields separated by the colon (':') character. The join field number is 3 for **filea** and 1 (by default) for **fileb**. The selected five fields are produced in the output.

See Also

awk, comm, commands, sort, uniq

K

kermit — Command

Inter-system communication and file transfer
kermit c[belL *baud esc line*]
kermit r[bdfhilLt *baud line*]
kermit s[abdfhilLmtx *baud line*] *file* ...

kermit allows the user to communicate with a remote computer system and to transfer files between the local and remote systems. **kermit** can transfer ASCII or binary files of any length in either direction. The two computers must be able to contact each other, such as through a serial line or by modem over a telephone line, and both systems must have **kermit** available. The user must have login privileges on both systems and appropriate permissions in directories used for file transfer.

The **kermit** command line specifies a *mode*, followed without intervening spaces by optional *flags*, perhaps followed by additional arguments and *files*. The three possible *modes* are as follows:

c Connect the two systems so they can communicate
r Receive files from the other system
s Send each *file* to the other system.

kermit normally uses a default communication line at a default baud rate; the defaults vary from system to system. **kermit** normally strips leading directory information from the path name of each *file* it sends and converts the name to upper case; it converts the file name to lower case when receiving.

The following *flags* modify **kermit**'s normal behavior.

a Specify complete path names for sending and receiving files. Used only with **s** mode. This flag requires file names in pairs: first gives the file to be sent, the second the receiving file. For example, the command

```
kermit sa /usr/joe/stuff.c /usr/tom/src/thing.c
```

sends the file **/usr/joe/stuff.c** but specifies its name as **/usr/tom/src/thing.c** for the receiving system. The target directory must exist on the receiving system. The **a** flag implies the use of the **f** and **x** flags described below.

b *baud* Set the baud rate of the port to *baud*.

d Debug mode. Tell **kermit** to print messages that describe its actions. Message appear on the standard output, not the standard error.

e *esc* Change the escape character from the default '^' to *esc*; used only with **c** mode. The escape character marks commands to **kermit c** while it is running, as described below.

f Suppress conversion of the case of file names.

h Host mode. Tell **kermit** to use the same line for file transfer and for communication; used with either **r** or **s** mode on the remote system only. When invoked with the **h** flag, **kermit** resets the line modes properly when it completes a file transfer. If you do not use the **h** flag, **kermit** will probably leave the remote system line in raw, no-echo mode.

i Image mode. Tell **kermit** to send a full eight-bit byte for each character; this is necessary to transfer binary (non-ASCII) files. If you use **i** flag when sending, also use it on the receiving system.

l *line* Use *line*. For example, the command

```
kermit clb /dev/tty50 1200
```

tells **kermit** to use line tty50 at 1200 baud instead of the default line and baud rate.

L Log all **kermit** commands into file **Log**.

m Macintosh mode. Necessary when sending files to an Apple Macintosh; used only with **s** mode.

t Tymnet mode. Allows Tymnet to keep up with file transmission.

x Allows the specification of a complete pathname for the receiving file; used only with **s** mode. For example, the command

```
kermit sx mydir/stuff
```

sends the file **mydir/stuff** to **mydir/stuff** on the receiving system. The target directory must exist on the receiving system and the user must have write permission in it.

kermit c recognizes two escape sequences. The default escape character '^' can be changed with the **e** flag, as noted above.

^c Exit from **kermit** and break the connection between the two systems. This notation does *not* mean **<ctrl-C>**; rather, you must literally type the escape character (by default, a carat '^') and then the letter 'c'.

^s Suspend **kermit** on the host system but do not hang up the line.

Unlike some file transfer protocols, **kermit** requires that you invoke it on both the sending and receiving systems to transfer a file. As shown in the example below, you normally use **kermit c** to connect to the remote system, invoke **kermit** with the **h** flag in either send or receive mode on the remote system only, type "^s" to suspend the local **kermit c**, and finally invoke **kermit** in receive or send mode on the local system.

The following example demonstrates the use of **kermit**. The example assumes the user is already logged in on the local system. The communication line is **/dev/com2** and runs at 2400 baud. The user wants to transfer **locfile** to the remote system and **remfile** from the remote system. System names are in *italics* on the left, user input is in Roman, system responses are in **bold**, and remarks are in parentheses.

local	kermit clb /dev/com2 2400	(connect to remote system)
local	**kermit: connected...**	(type a carriage return)
remote	**Coherent login:**	(perform login procedure)
remote	kermit shi remfile	(send from remote)
remote	**)S~_@X#T**	(part of protocol, ignore)
remote	^s	(suspend local **kermit**)

kermit

local	**kermit: suspended.**	
local	kermit rilb /dev/com2 2400	(receive on local)
local	**kermit: Receiving REMFILE as remfile**	
local	**kermit: done.**	
local	kermit clb /dev/com2 2400	(connect again)
remote	kermit rhi	(receive on remote)
remote	^s	(suspend local **kermit**)
local	**kermit: suspended.**	
local	kermit silb /dev/com2 2400 locfile	(send from local)
local	**kermit: Sending locfile as LOCFILE**	
local	**kermit: done.**	
local	kermit clb /dev/com2 2400	(connect again)
remote	<ctrl-D>	(log off the remote system)
remote	**Coherent login:**	
remote	^c	(disconnect local **kermit**)
local	**kermit: disconnected.**	

Problems Connecting to the Modem

Some users occasionally experience problems in having **kermit** talk to their modems. The **kermit** utility requires that the serial port it uses for communications *not* be enabled for logins. If you wish to use **kermit** on your modem line, you must perform the following steps. Note that comments are shown in italic.

`who`	*(make sure nobody is logged in on the modem)*
`/etc/disable` *port*	*(where "port" is the modem port)*
`kermit` *options*	*(invoke kermit as needed)*
. . .	
(disconnect via kermit)	
`/etc/enable` *port*	*(re-enable modem port for logins)*

See Also

commands, UUCP

Kermit: A file-transfer protocol for universities, *BYTE*, June 1984 pp. 255*ff*, July 1984 pp. 143*ff*

Diagnostics

kermit may print the following error messages:

Aborting with following error from remote host
: Problem appeared on receiving system.

Bad line speed
: Transmission was attempted at an illegal baud rate.

Cannot create *name*
: The receiving system cannot create *name*. Confirm that you have write permission on the receiving system.

Cannot open file *name*
: The sending system cannot open *name*. Either you do not have read permission on the sending system, or the file is not present in the named directory.

Cannot open *line*
: An incorrect *line* number was specified.

No line specified for connection
: The *line* argument missing after the **-l** option.

Receive failed
: The file being sent was not received; this could be due to any one of a number of reasons. Check that everything is functioning normally, and then try to send the file again.

Send failed
: The requested file was not sent.

Speed setting not implemented
: An unimplemented baud rate was selected for the **-b** option.

Yes, I'm still here still here'>=29
: The connect command was repeated.

Notes

If you type **kermit c** and get the message **kermit connected** but the remote system does not respond, check the line that connects the two systems and the ability of the remote system to accept a login on the line.

The file transfer protocol uses small (96-character) checksummed packets, with ACK/NAK responses from the receiving system. The timeout period is five seconds, and **kermit** does ten retries before it abandons an attempted file transfer.

The **kermit** protocol was developed at the Columbia University Center for Computing Activities. Tymnet is a trademark of Tymshare, Inc.

keyboard tables — Technical Information

How to write a keyboard table

The COHERENT device-driver **nkb** supports industry-standard 83-, 101-, and 102-key AT-protocol keyboards attached as the computer console.

nkb lets you define both the layout of the keyboard and the values returned by function keys. You can change layout and function-key bindings by using the special keyboard mapping programs kept in directory **/conf/kbd**. This directory contains the C source code for the mapping tables, as well as a **Makefile** that helps you rebuild the mapping programs.

Before you begin to write or modify an existing keyboard table, be sure to read throroughly this article and the Lexicon article on **nkb**. If you do not, you may foul up the keyboard so thoroughly that it will not work well enough for you to undo your mistake!

Operational Overview

The device driver **nkb** provides the system's portion of the interface to the console keyboard. It handles hardware-specific details, such as initializing the keyboard and internal state, handling keyboard interrupts, processing key scan codes, and queueing characters.

The user half of the keyboard interface is provided by a set of stand-alone utilities. With these, you can program the **nkb** driver via specialized **ioctl()** calls. These utilities differ from each other only in the keyboard binding or mapping tables each uses. You can re-construct the interface to the **nkb** driver by modifying a keyboard-mapping file and then using a support module to link that file to the driver.

The keyboard-mapping file is a C program that consists of initialized tables and strings. In addition, several header files provide the scan codes and other constants required for the key tables. This format makes the file easy to edit, and also lets you enter characters in several different formats.

The support module, in turn, performs several tasks. These include scanning the keyboard-mapping file for errors, reformatting the table for use by the device driver, and passing the reformatted table to the driver.

Key Mapping Files

By convention, directory **/conf/kbd** contains the keyboard-mapping files, executables, and a **Makefile** that you use to construct the executables from the corresponding source files.

A keyboard-mapping source file consists primarily of three data structures that you must modify to support a given keyboard mapping. The first, and simplest, of the structures is **tbl_name**. This is a character string that describes the keyboard. For example, the stock 101-key US AT keyboard mapping file **/conf/kbd/us.c** initializes this string to:

```
"U.S. AT keyboard table"
```

The second data structure, **kbtbl**, is an array of key-mapping entries. It has one entry (or row) for each possible key location. Each entry in this structure consists of 11 fields, which hold, respectively, the key number, nine possible mapping values, and a mode field. The following example is for physical key location 3 from key-mapping source file **/conf/kbd/belgian.c**:

```
{ K_3, 0x82, '2', none, none, 0x82, '2', '~', none, '~', O|T },
```

Field 1 contains the *scan code set 3* code value for the desired key. Header file **<sys/kbscan.h>** contains symbolic constants of the form **K_***nnn* that map the AT keyboard's *physical* key number *nnn* to the corresponding scan code set 3 value generated by the keyboard. In the above example, **K_3** corresponds to key location three.

Fields 2 through 10 contain the key mappings corresponding to the following shift states, as follows:

2	base or unshifted
3	**SHIFT**
4	**CONTROL**
5	**CONTROL+SHIFT**
6	**ALT**
7	**ALT+SHIFT**
8	**ALT+CONTROL**
9	**ALT+CONTROL+SHIFT**
10	**ALT_GRAPHIC**

For "regular" keys, the values for these nine fields are eight-bit characters; for "function" or "shift" keys, they are special values. The symbolic constant **none** indicates that you want no output when the key is pressed in the specified shift state.

In the case of a function key, the value specified is the number of the desired function key. Header file **<sys/kb.h>** defines a set of symbolic constants of the form **f***n*, where *n* is the desired function key number. You should use these constants; they will improve the readability of your code, and they will protect your keyboard mapping source files from any future changes in the structure of the keyboard driver.

In the case of a "shift" key, all nine entries must be identical and must consist of one of the

following symbolic constants: **scroll**, **num**, **caps**, **lalt**, **ralt**, **lshift**, **rshift**, **lctrl**, **rctrl**, or **altgr**. These are defined in the **<sys/kb.h>** header file. Note that 83-key XT-layout keyboards only have one "control" and "alt" key, so not all shift-key combinations may be possible on your target keyboard.

The last (11th) field in the key entry is the "mode" field. The following symbolic constants specify the mode of the current key:

C The **caps lock** key affects this key.

F The specified key is a "function" or special key. The value of all mapping entries must name function keys. See header file **<kb.h>** for a list of predefined function keys.

M Make: use this mode with keys that do not repeat. Note that accidentally using this mode with "shift" keys will stop you from being able to "unshift" upon releasing the key!

MB Make/Break: use this mode with "shift" keys.

N The **num lock** key affects this key.

O The specified key is "regular" and requires no special processing.

S The specified key is a "shift" or "lock" key. Note that all mapping entries for a given key must be identical for a "shift" or "lock" key to work correctly.

T Typematic: this type is usually associated with a "regular" key.

TMB Typematic/Make/Break.

The above example specifies a mode field of **O|T**, which corresponds to a "regular" key with Typematic repeat, and no special handling of the "lock" keys.

The last data structure, **funkey**, consists of an array of function-key initializers, one per function key. The initializers are simple quoted character strings delimited by either hexadecimal value **0xFF**, octal value **\377**, or symbolic constant **DELIM**. Note that any other value can be used as part of a function-key binding. Function keys are numbered starting at zero. By convention, function key 0, when enabled, reboots your computer. For traditional reasons, this function key is usually bound to the key sequence **<ctrl><alt><del>**.

Function keys are useful not only in the classic sense of the programmable function keys on the keyboard, but also as a general purpose mechanism for binding arbitrary length character sequences to a given key. For example, physical key location 16 is usually associated with the **<tab>** and **<back tab>** on the AT keyboard; and **/conf/kbd/us.c** sets the key mapping table entry for key 16 as follows:

```
{ K_16, f42, f43, none, none, f42, f43, none, none, none, F|T },
```

For traditional reasons, the **<back tab>** key outputs the sequence **<esc>[Z** whereas the **<tab>** key simply outputs the horizontal-tab character **<ctrl-I>**. Because at least one of the mapping values for this key is more than one character long, the key must be defined as a "function" key and all entries for the the key must correspond to function-key numbers. In this example, function key number 42 was chosen for **<tab>**, and function key number 43 was chosen for **<back tab>**. The constant **none** indicates that you want no output when the key is pressed in the specified shift state. The corresponding **funkey** initialization entries for function keys **f42** and **f43** are as follows:

```
/* 42 */      "\t\377",          /* Tab */
/* 43 */      "\033[Z\377",      /* Back Tab */
```

We strongly recommend that you comment your function-key bindings.

You can also change function-key bindings via the command **fnkey**. This command lets you temporarily alter one or more function-key mappings without changing your key-mapping sources.

Building New Binaries

After you have modified an existing keyboard-mapping table, use the following commands to rebuild the corresponding executables:

```
cd /conf/kbd
su root
make
```

If you have created a new keyboard mapping table, you must edit **/conf/kbd/Makefile**. Duplicate an existing entry from the **Makefile**, and change the duplicated name to match the name of your new keyboard-mapping table. After you have finished your editing, build an executable from your source file by simply executing the above series of commands.

To load your new keyboard table, simply type the name of the executable that corresponds to your keyboard-mapping file. For example, if you just built executable **french** from source file **french.c**, type the following command:

```
/conf/kbd/french
```

If the keyboard-support module finds an error, it will print an appropriate message. If it finds no errors, it will update the internal tables of the **nkb** keyboard driver, reprogram the keyboard, and print a message of the form:

```
Loaded French AT keyboard table
```

Examples

Prior to the release of the 101- and 102-key, enhanced-layout AT keyboards, the **<ctrl>** key was positioned to the left of 'A' key. Most terminals also locate the **<ctrl>** key there. The first example shows how to swap the left **<ctrl>** key and the **<caps-lock>** key on a 101- and 102-key keyboard. The **<caps-lock>** key is physical key 30, whereas the left **<ctrl>** key is physical key 58. Their respective entries in file **/conf/kbd/us.c** source file are as follows:

```
{ K_30, caps, caps, caps, caps, caps, caps, caps, caps, caps,  S|M },
{ K_58, lctrl,lctrl,lctrl,lctrl,lctrl,lctrl,lctrl,lctrl,lctrl, S|MB },
```

Note that the **<caps-lock>** key is defined with mode **M** as it is a "lock" key. The keyboard will interrupt only on key depressions, because releasing a "lock" key has no effect. The left **<ctrl>** key is defined with mode **MB** as it is a "shift" key. The keyboard generates an interrupt on both key depression and key release, because the driver must track the state of this key.

To swap the aforementioned keys, simply change all occurrences of **caps** to **lctrl** and vice-versa, as well as swapping the mode fields. After making the changes, the entries now appear as:

```
{ K_30, lctrl,lctrl,lctrl,lctrl,lctrl,lctrl,lctrl,lctrl,lctrl, S|MB },
{ K_58, caps, caps, caps, caps, caps, caps, caps, caps, caps,  S|M  },
```

The second example converts a 101- or 102-key keyboard table to support an XT-style 83-key keyboard layout. The following section summarizes the "typical" differences found when comparing the two keyboard layouts. Needless to say, given the extreme variety in keyboard designs, your mileage may vary.

Location	*101/102 Value*	*83-key Value*	*Comments*
14	**none**	Various	Keyboard-specific
30	**caps**	**lctrl**	
58	**lctrl**	**lalt**	
64	**rctrl**	**caps**	
65	**none**	**F2**	Function Key
66	**none**	**F4**	Function Key
67	**none**	**F6**	Function Key
68	**none**	**F8**	Function Key
69	**none**	**F10**	Function Key
70	**none**	**F1**	Function Key
71	**none**	**F3**	Function Key
72	**none**	**F5**	Function Key
73	**none**	**F7**	Function Key
74	**none**	**F9**	Function Key
90	**num**	**Esc**	
95	/	**num**	
100	*	**scroll**	
105	-	**none**	**<SysReq>** not used
106	+	*	
107	**none**	-	
108	**<Enter>**	+	
110	**esc**	**none**	Not on XT layout
112-123	**F1-F12**	**none**	Not on XT layout
124	**none**	**none**	**<PrtScr>** not used
125	**scroll**	**none**	Not on XT layout
126	**none**	**none**	**<Pause>** not used

See Also

device drivers, fnkey, nkb

Notes

Key 14, if used, varies considerably among keyboard models.

The location of the key that contains characters '\' and '|' varies among 101-key US-layout keyboards.

When designing keyboard tables for keyboards that use the **ALT_GRAPHIC** shift key, for reasons of backwards compatibility you should allow the use of combination shift **ALT+CTRL** as a synonym for **ALT_GRAPHIC**.

kill() — COHERENT System Call (libc)

Kill a system process

#include <signal.h>
kill(*pid, sig*)
int *pid*, *sig*;

kill() is the COHERENT system call that sends a signal to a process. *pid* is the process identifier of the process to be signalled, and *sig* identifies the signal to be sent, as set in the header file **signal.h**. This system call is most often used to kill processes, hence its name.

See the Lexicon article for **signal** for a table of the signals and what each means.

See Also

COHERENT system calls, signal(), signal.h

kill — Command

Signal a process
kill [- *signal*] *pid* ...

COHERENT assigns each active process a unique process id, or *pid*, and uses the pid to identify the process. **kill** sends *signal* to each *pid*. *signal* must be one of the numbers described in the header **<signal.h>** or **<sys/msig.h>**. The signal can be given by number or by name, as **defined** in these header files. By default, *signal* is **SIGTERM**, which terminates a given process.

If *pid* is zero, **kill** signals each process started by the user from the same tty.

The shell prints the process id of a process if the command is detached. The command **ps** prints a list of all active processes, with process ids and command line arguments.

A user can kill only the processes he owns; the superuser, however, can kill anything. A process cannot ignore or catch **SIGKILL**.

See the Lexicon article for **signal** for a table of the signals and what each means.

Files

<sys/msig.h> — Machine-dependent signal numbers
<signal.h> — Machine invariant signal numbers

See Also

commands, getpid(), init, kill(), ksh, ps, sh, signal()

ksh — Command

The Korn shell
ksh *token* ...

The COHERENT system offers two command interpreters: **sh**, the Bourne shell; and **ksh**, the Korn shell. **sh** is the default COHERENT command interpreter. The shell tutorial included in this manual describes the Bourne shell in detail.

This article describes **ksh**, the Korn shell. **ksh** is a superset of the Bourne shell, and contains many features that you may well find useful. These include MicroEMACS-style editing of command lines; command hashing; a full-featured aliasing feature; and a job-control facility.

Invoking ksh

To invoke **ksh** from within the Bourne shell, simply type **ksh** at the command-line prompt. To use **ksh** as your default shell, instead of **sh**, append the command **/usr/bin/ksh** to the end of your entry in the file **/etc/passwd**. (See the Lexicon entry for **passwd** for more information on this file.)

You can invoke **ksh** with one or more built-in options; these are described below.

Commands

A *command* consists of one or more *tokens*. A *token* is a string of text characters (i.e., one or more alphabetic characters, punctuation marks, and numerals) delineated by spaces, tabs, or newlines.

A *simple command* consists of the command's name, followed by zero or more tokens that represent arguments to the command, names of files, or shell operators. A *complex command* will use shell constructs to execute one or more commands conditionally. In effect, a complex command is a mini-program that is written in the shell's programming language and interpreted by **ksh**.

Shell Operators

The shell includes a number of operators that form pipes, redirect input and output to commands, and let you define conditions under which commands are executed.

command **|** *command*

The *pipe* operator: let the output of one command serve as the input to a second. You can combine commands with '|' to form *pipelines*. A pipeline passes the standard output of the first (leftmost) command to the standard input of the second command. For example, in the pipeline

```
sort customers | uniq | more
```

ksh invokes **sort** to sort the contents of file **customers**. It pipes the output of **sort** to the command **uniq**, which outputs one unique copy of the text that is input into it. **ksh** then pipes the output of **uniq** to the command **more**, which displays it on your terminal one screenful at a time. Note that under COHERENT, unlike MS-DOS, pipes are executed concurrently: that is, **sort** does not have to finish its work before **uniq** and **more** can begin to receive input and get to work.

command **;** *command*

Execute commands on a command line sequentially. The command to the left of the ';' executes to completion; then the command to the right of it executes. For example, in the command line

```
a | b ; c | d
```

first execute the pipeline **a | b** then, when **a** and **b** complete, execute the pipeline **c | d**.

command **&**

Execute a command in the background. This operator must follow the command, not precede it. It prints the process identifier of the command on the standard output, so you can use the **kill** command to kill that process should something go wrong. This operator lets you execute more than one command simultaneously. For example, the command

```
/etc/fdformat -v /dev/fha0 &
```

formats a high-density, 5.25-inch floppy disk in drive 0 (that is, drive A); but while the disk is being formatted, **ksh** returns the command line prompt so you can immediately enter another command and begin to work. If you did not use the '&' in this command, you would have to wait until formatting was finished before you could enter another command.

ksh also prints a message on your terminal when a command that you are running in the background finishes processing. It does not check these "child" processes very often, however, so a command may have finished some time before **ksh** informs you of the fact. See the Lexicon article for the command **ps** for information

on all processes; also see the description of the built-in command **jobs**, below.

command **&&** *command*

Execute a command upon success. **ksh** executes the command that follows the token '&&' only if the command that precedes it returns a zero exit status, which signifies success. For example, the command

```
cd /etc
fdformat -v /dev/fha0 && badscan -o proto /dev/fha0 2400
```

formats a floppy disk, as described above. If the format was successful, it then invokes the command **badscan** to scan the disk for bad blocks; if it was not successful, however, it does nothing.

command **||** *command*

Execute a command upon failure. This is identical to operator '&&', except that the second command is executed if the first returns a non-zero status, which signifies failure. For example, the command

```
/etc/fdformat -v /dev/fha0 || echo "Format failed!"
```

formats a floppy disk. If formatting failed, it echoes the message **Format failed!** on your terminal; however, if formatting succeeds, it does nothing.

Note that the tokens newline, ';' and '&' bind less tightly than '&&' and '||'. **ksh** parses command lines from left to right if separators bind equally.

>*file* Redirect standard output. The *standard input*, *standard output*, and *standard error* streams are normally connected to the terminal. A pipeline attaches the output of one command to the input of another command. In addition, **ksh** includes a set of operators that redirect input and output into files rather than other commands.

The operator '>' redirects output into a file. For example, the command

```
sort customers >customers.sort
```

sorts file **customers** and writes the sorted output into file **customers.sort**. It creates **customers.sort** if it does not exist, and destroys its previous contents if it does exist.

>>*file* Redirect output into a file, and append. If the file does not exist, this operator creates it; however, if the file already exists, this operator appends the output to that file's contents rather than destroying those contents. For example, the command

```
sort customers.now | uniq >>customers.all
```

sorts file **customers.now**, pipes its output to command **uniq**, which throws away duplicate lines of input, and appends the results to file **customers.all**.

<*file* Redirect input. Here, **ksh** reads the contents of a file and processes them as if you had typed them from your keyboard. For example, the command

```
ed textfile <edit.script
```

invokes the line-editor **ed** to edit **textfile**; however, instead of reading editing commands from your keyboard, the shell passes **ed** the contents of **edit.script**. This command would let you prepare an editing script that you could execute repeatedly upon files rather than having to type the same commands over and over.

<< token

Prepare a "here document". This operator tells **ksh** to accept standard input from the shell input until it reads a line that contains only *token*. For example, the command

```
cat >FOO <<\!
     Here is some text.
!
```

redirects all text between '<<\!' and '!' to the **cat** command. The '>' in turn redirects the output of **cat** into file **FOO**. **ksh** performs parameter substitution on the here document unless the leading *token* is quoted; parameter substitution and quoting are described below.

command **2>** *file*

Redirect the standard error stream into a file. For example, the command

```
nroff -ms textfile >textfile.p 2>textfile.err
```

invokes the command **nroff** to format the contents of **textfile**. It redirects the output of **nroff** (i.e., the standard output) into **textfile.p**; it also redirects any error messages that **nroff** may generate into file **textfile.err**.

Note in passing that a command may use up to 20 streams. By default, stream 0 is the standard input; stream 1 is the standard output; and stream 2 is the standard error. **ksh** lets you redirect any of these streams individually into files, or combine streams into each other.

<&*n*

ksh can redirect the standard input and output to duplicate other file descriptors. (See the Lexicon article **file descriptor** for details on what these are.) This operator duplicates the standard input from file descriptor *n*.

>&*n*

Duplicate the standard output from file descriptor *n*. For example,

```
2>&1
```

redirects file descriptor 2 (the standard error) to file descriptor 1 (the standard output).

Note that each command executed as a foreground process inherits the file descriptors and signal traps (described below) of the invoking shell, modified by any specified redirection. Background processes take input from the null device **/dev/null** (unless redirected), and ignore interrupt and quit signals.

File-Name Patterns

The shell interprets an input token that contain any of the special characters '?', '*', or '[' as a file name *pattern*.

?

Match any single character except newline. For example, the command

```
ls name?
```

will print the name of any file that consists of the string **name** plus any one character. If **name** is followed by no characters, or is followed by two or more characters, it will not be printed.

*****

Match a string of non-newline characters of any length (including zero).

```
ls name*
```

prints the name of any file that begins with the string **name**, regardless of whether it is

followed by any other characters. Likewise, the command

```
ls name?*
```

prints the name of any file that consists of the string **name** followed by at least one character. Unlike **name***, the token **name?*** must be followed by at least one character before it will be printed.

~name
Replace the name of user *name* with his **$HOME** directory. For example, the command

```
ls -l ~norm/src
```

lists the contents of the *src* subdirectory located under the **$HOME** directory for user **norm**. This spares you from having to know where a given user's HOME directory is located.

[!*xyz*]
Exclude characters *xyz* from the string search. For example, the command

```
ls [!abc]*
```

prints all files in the current directory except those that begin with **a**, **b**, or **c**.

[*C-c*]
Enclose alternatives to match a single character. A hyphen '-' indicates a range of characters. For example, the command

```
ls name[ABC]
```

will print the names of files **nameA**, **nameB**, and **nameC** (assuming, of course, that those files exist in the current directory). The command

```
ls name[A-K]
```

prints the names of files **nameA** through **nameK** (again, assuming that they exist in the current directory).

When **ksh** reads a token that contains one of the above characters, it replaces the token in the command line with an alphabetized list of file names that match the pattern. If it finds no matches, it passes the token unchanged to the command. For example, when you enter the command

```
ls name[ABC]
```

ksh replaces the token **name[ABC]** with **nameA**, **nameB**, and **nameC** (again, if they exist in the current directory), so the command now reads:

```
ls nameA nameB nameC
```

It then passes this second, transformed version of the command line to the command **ls**.

Note that the slash '/' and leading period '.' must be matched explicitly in a pattern. The slash, of course, separates the elements of a path name; while a period at the begin of a file name usually (but not always) indicates that that file has special significance.

Quoting Text

From time to time, you will want to "turn off" the special meaning of characters. For example, you may wish to pass a token that contains a literal asterisk to a command; to do so, you need a way to tell **ksh** not to expand the token into a list of file names. Therefore, **ksh** includes the **quotation operators** '\', '"', and "'; these "turn off" (or *quote*) the special meaning of operators.

The backslash '\' quotes the following character. For example, the command

```
ls name\*
```

lists a file named **name***, and no other.

The shell ignores a backslash immediately followed by a newline, called a *concealed newline*. This lets you give more arguments to a command than will fit on one line. For example, the command

```
cc -o output file1.c file2.c file3.c \
      file4.c file5.c file19.c
```

invokes the C compiler **cc** to compile a set of C source files, the names of which extend over more than one line of input. You will find this to be extremely helpful, especially when you write scripts and **makefiles**, to help you write neat, easily read commands.

A pair of apostrophes '' prevents interpretation of any enclosed special characters. For example, the command

```
find . -name '*.c' -print
```

finds and prints the name of any C-source file in the current directory and any subdirectory. The command **find** interprets the '*' internally; therefore, you want to suppress the shell's expansion of that operator, which is accomplished by enclosing that token between apostrophes.

A pair of quotation marks "" has the same effect. Unlike apostrophes, however, **ksh** will perform parameter substitution and command-output substitution (described below) within quotation marks. Note that everything between quotation marks will be a single argument, even if there are spaces between the tokens. For example, the command

```
grep "x y" *.c
```

calls the string-search command **grep** to look for the string **x<space>y**.

Scripts

Shell commands can be stored in a file, or *script*. The command

```
ksh script [ parameter ... ]
```

executes the commands in *script* with a new subshell **ksh**. Each *parameter* is a value for a positional parameter, as described below. If you have used the command **chmod** to make *script* executable, you may omit the **ksh** command.

Parameters of the form '$*n*' represent command-line arguments within a script. *n* can range from zero through nine; **$0** always gives the name of the script. These parameters are also called *positional parameters*.

If no corresponding parameter is given on the command line, the shell substitutes the null string for that parameter. For example, if the script **format** contains the following line:

```
nroff -ms $1 >$1.out
```

then invoking **format** with the command line:

```
format mytext
```

invokes the command **nroff** to format the contents of **mytext**, and writes the output into file **mytext.out**. If, however, you invoke this command with the command line

```
format mytext yourtext
```

the script will format **mytext** but ignore **yourtext** altogether.

Reference **$*** represents all command-line arguments. If, for example, we change the contents of script **format** to read

```
nroff -ms $* >$1.out
```

then the command

```
format mytext yourtext
```

will invoke **nroff** to format the contents of **mytext** and **yourtext**, and write the output into file **mytext.out**.

Commands in a script can also be executed with the . (dot) command. It resembles the **ksh** command, but the current shell executes the script commands without creating a new subshell or a new environment; therefore, you cannot use command-line arguments.

Variables

Shell variables are names that can be assigned string values on a command line, in the form

name=value

The name must begin with a letter, and can contain letters, digits, and underscores '_'. Note that no white space can appear around the '=', or the assignment will not work.

In shell input, '$*name*' or '${*name*}' represents the value of the variable. For example:

```
TEXT=mytext

nroff -ms $TEXT >$TEXT.out
```

Here, **ksh** expands **$TEXT** before it executes the **nroff** command. This technique is very useful in large, complex scripts: by using variables, you can change the behavior of the script by editing one line, rather than having to edit numerous variables throughout the script.

Note that if an assignment precedes a command on the same command line, the effect of the assignment is local to that command; otherwise, the effect is permanent. For example,

```
kp=one testproc
```

assigns variable **kp** the value **one** only for the execution of the script **testproc**.

ksh sets the following variables by default:

The number of actual positional parameters given to the current command.

@ The list of positional parameters "$1 $2 ...".

***** The list of positional parameters "$1" "$2" ... (the same as '$@' unless some parameters are quoted).

- Options set in the invocation of the shell or by the **set** command.

? The exit status returned by the last command.

! The process number of the last command invoked with '&'.

$ The process number of the current shell.

Environmental Variables

ksh references the following environmental variables:

CWD Current working directory: this is the name of the directory in which you are now working.

ENV If this variable is set at start-up, after all **profile** files have been executed, the expanded value is used as the shell's start-up file. It typically defines functions and aliases.

FCEDIT This sets the editor used by the command **fc**.

HOME Initial working directory; usually specified in the password file **/etc/passwd**.

IFS Delimiters for tokens; by default space, tab, and newline.

KSH_VERSION
The current version of the Korn shell that you are using.

MAIL Checked at intervals specified by environmental variable **MAILCHECK**. If file specified by this variable is new since last checked, the shell prints "You have mail." on the user's terminal. If the file has increased in size since the last check, the shell prints "You have new mail." on the user's terminal.

MAILCHECK
Specifies the number of seconds between checking for new mail. If not specified, **MAILCHECK** defaults to 60 seconds.

OLDPWD The prior working directory, if any.

PATH Colon-separated list of directories searched for commands.

PS1 First prompt string, usually '$'. Note that in this variable and **PS2**, **ksh** expands the symbol ! into the current number of the command line. For example, the prompt **ksh !>** prints the prompt **ksh** *NN*> with every command, where *NN* is the number of the current command. This is useful when you have enabled the history feature, as described below.

To print a prompt that includes your local site name, include the variable **$PWD** (described below) in the definition of **PS1**. For example,

```
PS1='$PWD>'
```

prints the current directory as your prompt, just like MS-DOS does. To include your system's name, read the contents of file **/etc/uucpname**, as follows:

```
SITEE ='cat/etc/uucpname'
PS1='$SITE !! $PWD>'
```

This form of the prompt is quite useful when you are working on networked machines and may not always be sure just what system you are working on. Note that two exclamation points are necessary; as noted above, **ksh** expands one '!' into the number of the current command.

Finally, to include the command number with site name and current directory, do the following:

```
SITE='cat /etc/uucpname'
```

```
PS1='$SITE !! $PWD !>'
```

This will give you a very long prompt, but one with much information in it.

PS2 Second prompt string, usually '>'. **ksh** prints it when it expects more input, such as when an open quotation-mark has been typed but a close quotation-mark has not been typed, or within a shell construct.

PWD The present working directory, i.e., the directory within which you are now working.

SECONDS
The number of seconds since the current shell was started.

SHELL The full path name of the shell that you are now executing.

TERM The name of the type of terminal you are now using, as used by various programs for reading the file **/etc/termcap**.

TIMEZONE
The current timezone you are located in, as set in your **.profile**. This is an interesting and powerful variable; see its entry in the Lexicon for details.

USER The login-identifier of the user, i.e., you.

The following special forms substitute parameters conditionally:

${*name-token***}**
Substitite *name* if it is set; if it is not, substitute *token*.

${*name=token***}**
Substitute *name* if it is set; if it is not set, substitute *token* and set *name* to equal *token*.

${*name+token***}**
Substitute *token* if *name* is set.

${*name?token***}**
Substitute *name* if it is set; if it is not, print *token* and exit from the shell.

Command Output Substitution

ksh can use the output of a command as shell input (as command arguments, for example) by enclosing the command in grave characters ` `. For example, to list the contents of the directories named in file **dirs**, use the command

```
ls -l `cat dirs`
```

Constructs

ksh lets you control the execution of programs through the following constructs. It recognizes a construct only if it occurs unquoted as the first token of a command. This implies that a separator must precede each reserved word in the following constructs; for example, newline or ';' must precede **do** in the **for** construct.

break *[n]*
Exit from **for**, **until**, or **while**. If *n* is given, exit from *n* levels.

case *token* **in** [*pattern [| pattern]* ...) *sequence*;;] ... **esac**
Check *token* against each *pattern*, and execute *sequence* associated with the first matching *pattern*.

continue *[n]*
Branch to the end of the *n*th enclosing **for**, **until**, or **while** construct.

for *name [* **in** *token ...]* **do** *sequence* **done**
Execute *sequence* once for each *token*. On each iteration, *name* takes the value of the next *token*. If the **in** clause is omitted, **$@** is assumed. For example, to list all files ending with **.c**:

```
for i in *.c
do
      cat $i
done
```

if *seq1* **then** *seq2 [* **elif** *seq3* **then** *seq4] ... [* **else** *seq5]* **fi**
Execute *seq1*. If the exit status is zero, execute *seq2*; if not, execute the optional *seq3* if given. If the exit status of *seq3* is zero, then execute *seq4*, and so on. If the exit status of all tested sequences is nonzero, execute *seq5*.

time *sequence*
Time how long it takes *sequence* to execute. When *sequence* has finished exeucting, the time is displayed on the standard output.

while *sequence1 [* **do** *sequence2]* **done**
Execute *sequence2* as long as the execution of *sequence1* results in an exit status of zero.

(*sequence*)
Execute *sequence* within a subshell. This allows *sequence* to change the current directory, for example, and not affect the enclosing environment.

{*sequence*}
Braces simply enclose a *sequence*.

Built-in Commands

ksh executes most commands via the **fork** system call, which creates a new process. See the Lexicon articles on **fork()** and **exec** for details on these calls. **ksh** also has the following commands built into itself.

. *script* Read and execute commands from *script*. Positional parameters are not allowed. **ksh** searches the directories named in the environmental variable **PATH** to find the given *script*.

: *[token ...]*
A colon ':' indicates a "partial comment". **ksh** normally ignores all commands on a line that begins with a colon, except for redirection and such symbols as **$**, **{**, **?**, etc.

A complete comment: if **#** is the first character on a line, **ksh** ignores all text that follows on that line.

alias *[-***d***] [name=value ...]*
When called without arguments, **alias** prints all aliases and their values. When called with a *name* but no associated value, then it prints the value of *name*. When called with a *name* and *value* combination, it associated *value* with *name*.

For example, the command

```
alias logout='exit'
```

binds the token **logout** to the command **exit**: hereafter, whenever you type **logout**, it will be as if you typed the **exit** command.

The **-d** option creates an alias for a directory.

ksh has a number of aliases set by default. See the section **Aliases**, below, for details.

bind *[-m] [key_sequence=binding_name ...]*
When called without arguments, list the current set of key bindings for MicroEMACS-style editing of command lines. When called with arguments, bind the *key_sequence* to *binding_name*.

For example, the command

```
bind '^[^H'=delete-word-backward
```

binds the editing command **delete-word-backward** to the key sequence **<esc><backspace>**. Note that the carat characters in this command are literally that, not the shell's representation of a literal **<esc>** or **<backspace>** character.

When called with the **-m** option, bind more than one *binding_name* to a given *key_sequence*. This lets you build keyboard macros, to perform complex editing tasks with one or two keystrokes.

See the section on **Command-line Editing**, below, for details.

builtin *command*
Execute *command* as a built-in command.

cd *dir* Change the working directory to *dir*. If no argument is given, change to the home directory as set by the environmental variable **HOME**. When invoked, it also changes the environmental variables **PWD** and **OLDPWD**.

Using a hyphen '-' as the argument causes **ksh** to change to the previous directory, i.e., the one indicated by shell variable **OLDPWD**. In effect, this swaps **OLDPWD** and **PWD**, thus allowing you to flop back and forth easily between two directories.

echo *token ...*
Echo *token* onto the standard output. **ksh** replaces the command **echo** with the alias **echo='print'**.

eval *[token ...]*
Evaluate each *token* and treat the result as shell input.

exec *[command]*
Execute *command* directly rather than as a subprocess. This terminates the current shell.

exit *[status]*
Set the exit status to *status*, if given, then terminate; otherwise, the previous status is used.

export *[name ...]*
ksh executes each command in an *environment*, which is essentially a set of shell variable names and corresponding string values. It inherits an environment when invoked, and normally it passes the same environment to each command it invokes. **export** specifies that the shell should pass the modified value of each given *name* to the environment of subsequent commands. When no *name* is given, **ksh** prints the name of each variable marked for export.

export *VARIABLE=value*

This form of the **export** command sets *VARIABLE* to *value*, and exports it. Thus, the command

```
export FOO=bar
```

is equivalent to the commands:

```
FOO=bar
export FOO
```

fc *[-l] [-n] [first [last]]*

Draw the previously executed commands *first* through *last* back for manipulation and possible execution. *first* and *last* can be referenced either by their history numbers, or by a string with which the command in question begins. Normally, the commands are pulled into an editor for manipulation before they are executed; the editor is defined by the environmental variable **FCEDIT** (default, **ed**). The commands in question are executed as soon as you exit from the editor. Option **-l** lists the command(s) on **stdout**, and so suppresses the editing feature. Option **-n** inhibits the default history numbers.

fc -s *[old=new] [command]*

Re-execute *command* after substituting string *new* for *old*.

function *funcname { script }*

Define function **funcname** for the shell to execute. For example the following defines function **get_name** for the shell:

```
function get_name {
     echo -n Please enter your name ...
     read name
     return 0
}
```

When **ksh** encounters **get_name**, it runs the above-defined function, rather than trying to find **get_name** on the disk. Note that the return status can be any valid status and can be checked in the code that called **get_name** by reading the shell variable **$?** (described above), or by using the function as the argument to an **if** statement. This allows you to build constructs like the following:

```
if get_name; then
     do_something
else
     do_something_else
fi
```

To list all defined functions, type the alias **functions**. To receive detailed information on a defined function, use the command **type***name* where *name* is the name of the function in which you are interested.

getopts *optstring name [arg ...]*

Parse the *args* to *command*. See the Lexicon entry for **getopts** for details.

hash *[-r] [name ...]*

When called without arguments, **hash** lists the path names of all hashed commands. When called with *name* **hash** check to see if it is an executable command, and if so adds it to the shell's hash list. The **-r** option removes *name* from the hash list.

kill *[-l] [signal] process ...*
Send *signal* to *process*. The default signal is **TERM**, which terminates the process. *signal* may either be a number or a mnemonic as **#define**d in header file **<signal.h>**. When called with the **-l** option, it lists all known types of signals. See the Lexicon entry for **kill** for details.

let *[expression]*
Evaluate each *expression*. This command returns zero if *expression* evaluates to non-zero (i.e., fails), and returns non-zero if it evalutes to zero (i.e., succeeds). This is useful for evaluating expressions before actually executing them.

print *[-nreu*n*] [argument ...]*
Print each *argument* on the standard output, separated by spaces and terminated with a newline. Option **-n** suppresses printing of the newline. Option **-u***n* redirects output from the standard output to file descriptor *n*.

Note that each *argument* can contain the following standard C escape characters: **\b**, **\f**, **\n**, **\r**, **\v**, and **\###**. See the Lexicon article on **C Language** for details each character's meaning. The option **-r** inhibits this feature, and the **-e** option re-enables it.

read *name ...*
Read a line from the standard input and assign each token of the input to the corresponding shell variable *name*. If the input contains fewer tokens than the *name* list, assign the null string to extra variables. If the input contains more tokens, assign the last *name* the remainder of the input.

readonly *[name ...]*
Mark each shell variable *name* as a read-only variable. Subsequent assignments to read-only variables will not be permitted. With no arguments, print the name and value of each read-only variable.

return *[status]*
Return *status* to the parent process.

set *[-aefhkmnuvx [-o keyword] [name ...]]*
Set listed flag. The **-o** option sets *keyword*, where *keyword* is a shell option.

When used with one or more *names*, this command sets shell variables *name* to values of positional parameters beginning with **$1**.

For example, the command

```
set -h -o emacs ignoreeof
```

performs the following: turns on hashing for all commands, turns on MicroEMACS-style command-line editing, and turns off exiting upon EOF (that is, you must type **exit** to exit from the shell). **set** commands are especially useful when embedded in your **.profile**, where they can customize **ksh** to your preferences.

For details of this command, see its Lexicon entry.

shift Rename positional parameter **1** to current value of **$2**, and so on.

test *[option] [expression]*
Check *expression* for condition *option*. This is a useful and complex command, with more options than can be listed here. See its Lexicon entry for details.

times Print on the standard output a summary of processing time used by the current shell and all of its child processes.

trap *[command] [n ...]*
Execute *command* if **ksh** receives signal *n*. If *command* is omitted, reset traps to original values. To ignore a signal, pass null string as *command*. With *n* zero, execute *command* when the shell exits. With no arguments, print the current trap settings.

typeset *[-firx] [+firx] [name [=value] ...]*
When called without an argument, this command lists all variables and their attributes.

When called with an option but without a *name*, it lists all variables that have the specified attribute; **-** tells **typeset** to list the value of each variable and **+** tells it not to.

When called with one or more *names*, it gives *name* to the listed attribute. If *name* is associated with a *value*, **typeset** also assigns the *value* to it.

typeset recognizes the following attributes:

- **-i** Store variable's value as an integer
- **-f** List function instead of variable
- **-r** Make the variable read-only
- **-x** Export variable to the environment

umask *[nnn]*
Set user file creation mask to *nnn*. If no argument is given, print the current file creation mask.

unalias *[-d] name ...*
Remove the alias for each *name*. The **-d** option unaliases an alias for a directory.

wait *[pid]*
Hold execution of further commands until process *pid* terminates. If *pid* is omitted, wait for all child processes. If no children are active, this command finishes immediately.

whence *[-v] name ...*
List the type of command for each *name*. When called with the **-v** option, also list functions and aliases.

Aliases

ksh implements as aliases a number of commands that **sh** calls as separate executable programs. The **echo** alias, for instance, does everything that **/bin/echo** does, but **ksh** does not have to **fork()** and **exec()** simply to echo a token. Other aliases, like **pwd**, work by printing the contents of shell variables. The command **/bin/pwd** still works should you prefer it, but you must request it by its full path name should you not wish to use the much faster alias version.

ksh sets the following aliases by default. If you wish, you can use the built-in command **unalias** to make one or all of them go away.

```
echo=print
false=let
functions=typeset -f
history=fc -l
integer=typeset -i
```

login=exec login
newgrp=exec newgrp
pwd=print -r $PWD
r=fc -s
true=:
type=whence -v

The alias **history** is especially useful when you are using the Korn shell's history feature. When invoked with no argument, it prints the last 13 commands you typed. When invoked with one numeric argument, it lists the command that corresponds to that argument; for example

```
history 106
```

prints the 106th command you entered (assuming that you've entered that many). When used with two numeric arguments, it prints the range of commands between the two arguments; for example

```
history 10 99
```

prints the tenth through the 99th commands you entered.

Job Control

ksh lets you manipulate and monitor background jobs via its *job control* commands.

The following commands manipulate background jobs:

jobs Display information about all controlled jobs. Information is in the following format:

%num [+-] *pid status command*

where *num* indicates the job number, '+' indicates the current job, '-' indicates the previous job, *pid* is the job's process identifier, *status* shows the status of the job (e.g., Running, Done, Killed), and *command* is the command description. Note that **ksh** only checks for changes in job status when waiting for a command to complete.

kill *[-signal] pid ...*
Described above.

wait *[pid]*
Hold execution of further commands until process *pid* terminates. See its Lexicon entry for details.

The following '%' syntax can be used with the above commands:

%+ Select the current job.

%- Select the previous job.

%*num* Select the job with job number *num*.

%*string*
Select the most recently invoked job whose command begins with *string*.

%?*string*
Select the most recently invoked job whose command contains *string*.

Command-line Editing

One of the most useful features of **ksh** is its ability to remember commands that you have typed previously. You can interactively edit previously issued commands and re-issue them with just a few keystrokes.

You can recall commands and edit them using the **fc** command, described above. **ksh**, however, also has built into it a MicroEMACS editing feature that lets you recall and edit commands using MicroEMACS-style editing commands. When you have finished editing, simply typing **<enter>** dispatches the command for re-execution.

To turn on MicroEMACS editing, use the command

```
set -o emacs
```

The following table gives each editing command and its default keybinding. Note that you can replace any of the following keybindings by using the **bind** command, described above. Note, too, that not every command has a default keybinding. Those that do not have one are marked "None".

abort (**<ctrl-G>**)
: Abort the current input line or function.

auto-insert
: Insert text into the command line. This is the default for almost every key.

backward-char (**<ctrl-B>**)
: Move the cursor one character to the left.

backward-word (**<esc>B**)
: Move the cursor one word to the left. A **word** is defined as any cluster of characters delineated by any of the characters named in the environmental variable **IFS**: by default, **<space>**, **<tab>**, and **<newline>**.

beginning-of-line (**<ctrl-A>**)
: Move the cursor to the leftmost position (i.e., the beginning) of the line.

complete (**<esc><esc>**)
: Complete as much as is unique of the hashed command name or file name in which the cursor is positioned. If no unique command or file name is found, **ksh** beeps. Note that this command does nothing unless you have used the **set** command to turn on hashing.

complete-command (**<ctrl-X><esc>**)
: Automatically complete as much as is unique of the hashed command name. Like the **complete** command, above, except that file names are not expanded.

complete-file (**<ctrl-X><ctrl-X>**)
: Automatically complete as much as is unique of the file name. Like the **complete** command, above, except that commands are not expanded.

delete-char-backward (**<ctrl-H>**)
: Delete the character to the left of the cursor. Shift text to the left to fill the gap left by the deleted character.

delete-char-forward (**<ctrl-D>**)
: Delete the character upon which the cursor is positioned. Shift text to the left to fill the gap left by the deleted character.

delete-word-backward (**<ctrl-W>**)
Delete the word to the left of the cursor. Shift text to the left to fill the gap left by the deleted word.

delete-word-forward (**<esc>D**)
Delete the word to the right of the cursor. Shift text to the left to fill the gap left by the deleted word.

down-history (**<ctrl-N>**)
Scroll to the next command in the history buffer, if any.

end-of-line (**<ctrl-E>**)
Move the cursor to the rightmost position (i.e., the end) of the line.

eot (**<ctrl-_>**)
Send an EOT (end of transmission) signal to the shell. Normally, this is sent by **<ctrl-D>**, but MicroEMACS mode binds this keystroke to an editing command.

forward-char (**<ctrl-F>**)
Move the cursor one character to the right.

forward-word (**<esc>F**)
Move the cursor one word to the right.

kill-line (**<ctrl-U>**)
Delete (i.e., erase) this entire input line.

kill-to-eol (**<ctrl-K>**)
Kill the input line from where the cursor is positioned to the end of the line.

list (**<esc>?**)
Display a sorted listed of all hashed commands and file names that have been entered so far, and so lists the tokens that can be expanded with the **complete** commands, described above.

list-command (**<ctrl-X>?**)
List all hashed commands.

list-file (none)
List all files used in hashed commands so far.

newline (**<ctrl-J>** or **<ctrl-M>**)
Dispatch the current line to the shell for execution. The cursor need not be at the beginning or end of the line for this command to work correctly.

prefix-1 (**<esc>**)
Introduce a two-character command sequence.

prefix-2 (**<ctrl-X>**)
Introduce a two-character command sequence.

quote (**<ctrl-^>**)
Read the following character literally, rather than as an editing command.

redraw (**<ctrl-L>**)
Redisplay the prompt and the current command line. This is useful if the line is garbled due to, say, line noise when you are using a modem.

search-character (**<ctrl-]>**)
Search forward in the current command line for the next character typed.

search-history (**<ctrl-R>**)
Enter incremental-search mode and search backwards through the history buffer. **abort** aborts search and returns you to the line from which you began the search; **<esc>** ends searching and leaves you in the current line.

stuff (none)
Take a character that is bound to an editing command and "stuff" it back into the terminal input, so it can receive special treatment by the terminal handler.

stuff-reset (none)
"Stuffs" a character, then aborts input.

transpose-chars (**<ctrl-T>**)
Swap the character the cursor in on, with the character to its left.

up-history (**<ctrl-P>**)
Move to the previous line in the history buffer (if any).

yank (**<ctrl-Y>**)
Insert the most recently killed text back into the command string, at the point where the cursor is positioned.

yank-pop (**<esc>Y**)
Yank a string, then replace it within the "yank" buffer with the next most previously killed string.

Example

The following C code creates a program called **splurt.c**. It demonstrates numbered redirection of **ksh**, by writing to five streams without opening them. Compile it with the command:

```
cc -o splurt splurt.c
```

To call it from the command line, you could type a command of the form:

```
splurt     3> splurt3 4> splurt4 5> splurt5 6> splurt6 7> splurt7
```

This will redirect the **splurt**'s output into files **splurt3** through **splurt7**.

```
#include <stdio.h>
main()
{
     int i;
     char buf[50];

     for(i = 3; i < 8; i++) {
          sprintf(buf, "For fd %d\n", i);
          write(i, buf, strlen(buf));
     }
}
```

Files

/etc/profile — System-wide initial commands
$HOME/.profile — User-specific initial commands
/dev/null — For background input

See Also

bind, commands, dup(), environ, exec, fork(), getopts, jobs, kill, login, newgrp, set, sh, signal(), test, wait

For a list of commands associated with **ksh**, see the **Shell Commands** section of the **Commands** Lexicon article.

Introduction to sh, the Bourne Shell, tutorial

Notes

Note that the queue of previously issued commands is stored in memory, not on disk.

This version of **ksh** offers a subset of the features of the Korn shell shipped with UNIX System V.2. It does *not* offer the following features:

- **vi**-style command-line editing.
- Command **fc -e**.
- Variables **RANDOM** and **PPID**.
- Variable arrays.
- Variable attributes other than integers.

The Mark Williams version of **ksh** is based on the public-domain version of the Korn shell, which in turn is based on the public-domain version of the seventh edition Bourne shell written by Charles Forsyth and modified by Eric Gisin, Ron Natalie, Arnold Robbins, Doug Gwyn, and Erik Baalbergen.

KSH_VERSION — Environmental Variable

List current version of Korn shell

The Korn shell stores its current version in environmental variable **KSH_VERSION**.

See Also

environmental variables, ksh

L

L-devices — File Format

Describe devices used by UUCP
/usr/lib/uucp/L-devices

The file **L-devices** describes the communication lines from your COHERENT system to other sites. It indicates whether a line is directly wired or under modem control, and it also gives the protocol needed to manipulate it. The command **uucico** reads the contents of **L-devices** before it attempts to transfer a file to or from a remote site.

Each entry in **L-devices** has five fields, each field being a string demarcated by one or more white-space characters. The fields are as follows:

1. Type
The first field defines the type of line. A line can be either of two types: **DIR** (for a directly wired line), or **ACU** (for a modem). Note the spelling: both entries must be entirely in upper case.

2. Line
The second field defines the serial line into which the device is plugged. This entry must specify the "local" COM device (e.g., **com3l** or **hs00**).

3. Disable
For modem devices (type **ACU**), the third field must give the remote variant for the entry in field 2 (e.g., for **com3l** the entry would be **com3r**, for **hs00** it is **hs00r**). The device named in this field must appear in the file **/etc/ttys**. For directly connected devices (type **DIR**), the device is the same as the entry in the second field.

4. Baud
The fourth field must give a legal baud rate for your modem, as specified in the entry in file **/etc/modemcap** for your modem.

5. Modem
Field 5 must correctly name your modem using an entry from **/etc/modemcap**.

The file **/etc/modemcap** contains descriptions for a number of popular modems, to spare you the trouble of typing control sequences for your modem. For a list of the modems described in this file, as well as available speeds, see the Lexicon entry for **modemcap**.

Example

The following entry in file **L-devices** specifies a 1200-baud Hayes (or Hayes-compatible) modem attached to serial port COM2:

```
ACU   com2l   com2r   1200   hayes
```

See Also

file formats, UUCP

l.out.h — Header File

Object file format
#include <l.out.h>

The header file **l.out.h** describes the format for the output of compilers, assemblers, and the linker.

The assembler outputs an unlinked object module, which must be bound with any required libraries (leaving no unresolved symbols) to produce an executable file, or *load module*. A call to one of the **exec** routines can then execute the load module directly.

The link module begins with a header, which gives global and size information about each segment. Segments of the indicated size follow the header in a fixed order. The header file **l.out.h** defines the header structure for the Z8000 and M68000 as follows:

```
struct    ldheader {
          short        l_magic;
          short        l_flag;
          short        l_machine;
          short        l_tbase;
          size_t       l_ssize[NLSEG];
          long         l_entry;
};
```

It describes the header structure for the i8086, i8088, i80286, and PDP-11 as follows:

```
struct    ldheader {
          int          l_magic;
          int          l_flag;
          int          l_machine;
          vaddr_t      l_entry;
          size_t       l_ssize[NLSEG];
};
```

l_magic is the magic number that identifies a link module; it always contains **L_MAGIC**. **l_flag** contains flags indicating the type of the object. **l_machine** is the processor identifier, as defined in the header file **mtype.h**. **l_tbase** is the start of the text segment. **l_entry** contains the machine address where execution of the module commences. **l_ssize** gives the size of each segment.

Files

l.out — Default load module name
<l.out.h>
<mtype.h> — Machine identifiers

See Also

as, cc, COHERENT system calls, core, exec, ld, mtype, nm

Notes

In the early releases of COHERENT, the header structure was defined only as shown above for i8086. It was changed to handle 32-bit addresses. In the future, the header structure defined above for Z8000 and M68000 machines will be implemented on i8086 and i80286 systems as well.

L.sys — File Format

Format for UUCP site descriptions
/usr/lib/uucp/L.sys

The file **L.sys** holds descriptions of remote sites that are accessed via UUCP. UUCP utilities read from this file the description of any system that you ask them to access. The superuser **root** can read and edit the contents of this file, both to update its contents and to add new descriptions.

Each line in **L.sys** is either a comment or a site descriptor. If a line begins with a pound sign ('#'), it is a comment; otherwise, it is treated as a site descriptor. Each site description consists of five or more fields, each field being demarcated by one or more white-space characters.

Site Description

The first five fields of a site description identify the site and how to contact it. These fields are as follows:

1. Remote system name

This names the remote system. In COHERENT versions 3.0.0 and 3.1.0, only the first seven characters are significant.

2. Legal call times

This entry specifies when the remote site may be called. There are several possible formats:

Never
day_list
day_and_time_list

Never means never call the remote site; use it only for sites that will only be calling you. *day_list* may be any of the following: **Any** (that is, call as soon as a file is queued for sending), **Wk** (for Monday through Friday), or one or more of **Su**, **Mo**, **Tu**, **We**, **Th**, **Fr**, or **Sa**, separated by commas (be sure not to use spaces here). A *day_and_time_list* is identical to a *day_list* but appends a *time* field to one or more of the days specified. The *time* field consists of two four-digit 24-hour times separated by a hyphen. The legal call time is any time at or after the first time and at or before the second time. If the first time is greater than the second time, then the valid calling times will be from midnight to the second time and from the first time through midnight. Omitting the *time* field permits calling at any time on the specified day. For example:

```
# never dial the site
Never
# dial the site whenever a file is queued
Any
# dial on Sunday, Monday and Tuesday, 2-5 AM
SuMoTu0200-0500
# Weekdays between 1-7 PM and all day on Saturday
Wk1300-1900,Sa
# Midnight Sunday through 2 AM and 11 PM through midnight
Su2300-0200
```

3. Device

This indicates the device on your computer via which UUCP is to contact the remote site. For sites accessed via a modem, use the entry **ACU**. For sites directly connected via a serial port, use the name of the port, e.g., **com3l**.

4. **Speed**

This gives the the baud rate at which UUCP is to call the remote system, e.g., 1200, 2400, or 9600. This speed must be valid according to the file **/etc/modemcap** for at least one modem described in the file **L-devices**.

5. Telephone number

This gives the string that UUCP is to send to the modem in order to call the remote site. This string may include special characters for your modem (e.g., some modems accept a comma if a pause is needed during dialing), but will usually be simply the number to dial, e.g. "17085590412". The string that is actually sent to the modem consists of the **cs** and **ds** strings from the modem's entry in **/etc/modemcap**, the telephone number, and finally the **de** and **ce** strings from **/etc/modemcap**.

Chat Script

The remaining fields in a descriptor form the "chat script", that is, the dialogue that your UUCP system must perform in order to log on to the remote computer. The chat script consists of strings of characters to be exchanged between the remote computer and your UUCP system; first comes an *expect_string* (the string that your system expects), followed by a *send_string* (the string to send in response to the *expect_string*). When calling a remote site, your computer waits for a carrier from the remote modem, then waits for the first *expect_string*, after which it sends the first *send_string*, etc.

Consider, for example, the remote system **bazooka**, which has a login prompt of **Coherent login:**; assume that your login is **howard**, the remote system prompts for passwords with **password:**, and your password is **r56d92**. The chat script for **bazooka** will read as follows:

```
ogin: howard word: r56d92
```

As you can see from the above example, an *expect_string* need contain only the last five characters of what the remote system sends.

An *expect_string* may be compound or simple. A *simple_expect_string* is either a sequence of characters (not including spaces, tabs, or hyphens) or a pair of quotation marks "". An empty pair of quotation marks tells UUCP not to wait for any incoming prompt, but go ahead and send the next *send_string* (i.e., expect a null string). A *compound_expect_string* is a sequence of fields separated by hyphens, in the format:

simple_expect_string-send_string-simple_expect_string-send_string...

A *compound_expect_string* has no spaces or tabs. If the first *simple_expect_string* is not received within 25 seconds, the first *send_string* after the hyphen is sent and the system waits for the second *simple_expect_string*; if the second *simple_expect_string* is not received after 25 seconds, the second *send_string* is sent, and so on. This syntax allows UUCP to use any number of alternate expect/send exchanges, rather than failing if it does not receive the first *expect_string*.

A *send_string* is the character sequence that UUCP sents to the remote site. Unless otherwise specified, UUCP sends a newline at the end of any *send_string*. As a special case of this, an empty alternate *send_string* tells UUCP to send a single newline to the remote site if the preceding *expect_string* is not received. For example

```
ogin:--ogin:
```

is a compound *expect_string* that tells UUCP to wait for the string **ogin:** from the remote site. If UUCP does not receive this string within the specified time limit, it sends a newline and again waits for **ogin:** from the remote site.

A *send_string* that consists of only two quotation marks '""' sends a carriage return and nothing else. Otherwise, it sends the text specified followed by a carriage return. You can embed the following escape sequences into a *send_string* to send special characters:

Notation	*Meaning*
EOT	Send an EOT character (\004)
BREAK	Send a break signal on the line
\b	Send a backspace
\c	Suppress the carriage return normally sent (can occur anywhere in send string)
\d	Delay for two seconds
\K	Send a break signal on the line
\n	Send a newline
\N	Send a NUL character (\000)
\p	Delay for one second
\r	Send a carriage return
\s	Send a space character
\t	Send a tab character
****	Send a backslash character
\xxx	Send the octal character specified

Limitations

L.sys has the following limitations:

- Site descriptors may not continue beyond one line.
- Line length cannot exceed 511 characters.
- No line may have more than 27 composite-expect/send pairs.
- In the COHERENT versions 3.0.0 and 3.1.0, there is no way to send a break signal to the remote modem. This feature will be added in a future release.

Example

The following **L.sys** entries are used to dial into the MWC UUCP BBS. The first entry corresponds to 2400 b.p.s. access and the second to 9600 b.p.s via a Telebit Trailblazer modem. Please note that in the example below, entries are continued over multiple lines; in the actual file, each entry must be on a single line, but the line may exceed 80 characters in length.

```
mwcbbs     Any ACU 2400 17085590412 \
     "" \r\d\r in:--in: nuucp word: public word: SERIALNUM
#mwcbbs    Any ACU 9600 17085590445 \
     FAST \r\d\r in:--in: nuucp word: public word: SERIALNUM
```

For further details on accessing the MWC BBS, refer to the COHERENT Release Notes.

See Also

file formats, L-devices, modemcap, Permissions, UUCP

l3tol() — General Function (libc)

Convert file system block number to long integer

l3tol(*lp, l3p, n*)
long **lp*;
char **l3p*;
unsigned *n*;

To conserve space inside i-nodes in COHERENT file systems, the system stores block addresses in three bytes. Programs that reference or maintain file systems use the functions **l3tol** and **ltol3** routines to convert between the three-byte representation and **long** numbers.

l3tol converts *n* three-byte block addresses at location *l3p* to an array of **long** integers at location *lp*.

See Also

canon.h, general functions, ltol3()

LASTERROR — Environmental Variable

Program that last generated an error
LASTERROR=*program name*

The environmental variable **LASTERROR** names the last program to have returned an error to the shell. For example, if you had used the command **set** with an incorrect number of arguments, it would have failed to run and would have exited with an error condition, and **LASTERROR** would read **LASTERROR=set**.

The command **help** reads **LASTERROR** to determine what the last program was for which you needed help. Thus, if you type **help** without an argument, it will return information about the program named in **LASTERROR**.

See Also

environmental variables

lc — Command

List/categorize files in a directory
lc [-1abcdfp] [*directory* **...]**

lc lists the names of the files in each *directory*, or the current directory if no *directory* is named. The files are categorized by type (files, directories, and so on) and listed in columns within each category.

The following options modify the output.

-1 List only one file name per line (do not print in columns).

-a List all file names, including '.' and '..'.

-b List block-special files only.

-c List character-special files only.

-d List directories only.

-f List regular files only.

-p List pipe files only.

See Also

commands, ls

ld — Command

Link relocatable object files
ld *[option ...] file ...*

A compiler translates a file of source code into a *relocatable object*. This relocatable object cannot be executed by itself, for calls to routines stored in libraries have not yet been resolved. **ld** combines, or *links*, relocatable object files with routines stored in libraries produced by the archiver **ar** to construct an executable file. For this reason, **ld** is sometimes called a *linker*, a *link editor*, or a *loader*.

ld scans its arguments in order and interprets each option as described below. Each non-option argument is either a relocatable object file, produced by **cc**, **as**, or **ld**, or a library archive produced by **ar**. It rejects all other arguments and prints a diagnostic message.

Each relocatable file argument is bound into the output file if its machine type matches the machine type of the first file so bound; if it does not, **ld** prints a diagnostic message. The symbol table of the file is merged into the output symbol table and the list of defined and undefined symbols updated appropriately. If the file redefines a symbol defined in an earlier bound module, the redefinition is reported and the link continues. The last such redefinition determines the value that the symbol will have in the output file, which may be acceptable but is probably an error.

Each library archive argument is searched only to resolve undefined references, i.e., if there are no undefined symbols, the linker goes to the next argument immediately. The library is searched from first module to last and any module that resolves one or more undefined symbols is bound into the output exactly as an explicitly named relocatable file is bound. The library is searched repeatedly until an entire scan adds nothing to the executable file.

The order of modules in a library is important in two respects: it will affect the time required to search the library, and, if more than one module resolves an undefined symbol, it can alter the set of library modules bound into the output.

A library will link faster if the undefined symbols in any given library module are resolved by a library module that comes later in the library. Thus, the low-level library modules, those with no undefined symbols, should come at the end of the library, whereas the higher-level modules, those with many undefined symbols, should come at the beginning. The library module **ranlib.sym**, which is maintained by the **ar s** modifier, provides **ld** with a compressed index to the symbols defined in the library. But even with the index, the library will link much faster if the modules occur in top-down rather than bottom-up order.

A library can be constructed to provide a type of "conditional" linking if alternate resolutions of undefined symbols are archived in a carefully thought-out order. For instance, **libc.a** contains the modules

finit.o
exit.o
_finish.o

in precisely the order given, though some other modules may intervene. **finit.o** contains most of the internals of the STDIO library, **exit.o** contains the **exit()** function, and **_finish.o** contains an empty version of **_finish()**, the function that **exit()** calls to close STDIO streams before process termination. If a program uses any STDIO routines, macros, or data, then **finit.o** will be bound into the output with its version of **finish()**. If a program uses no STDIO, then the "dummy" **_finish.o** will be bound into the output because it is the first module that defines **_finish()** that the linker encounters after **exit.o** adds the undefined reference. This saves approximately 3,000 bytes. To set the order of routines within a library, use the archiver **ar**.

The available options are as follows:

-d Define common regions even if relocation information is retained. By default, **ld** leaves common areas undefined if there are undefined symbols or if the **-r** option is specified.

-e *entry*
Specify the *entry* point of the output module, either as a symbol or as an absolute octal address.

-k[*system*]
Bind the output as a kernel process or linkable driver. The starting address depends on the target machine, and **ld** scans the *system* link file symbol table for symbols that are currently undefined. *system* defaults to **/coherent.**

-l *name* An abbreviation for the library **/lib/lib***name***.a** or **/usr/lib/lib***name***.a** if the first is not found.

-m This option tells **ld** to perform in-memory load if possible. This requires more memory, but is faster than using a buffer file.

-n Bind the output with separate shared and private segments, and with each starting on a hardware segment boundary, so that several processes can use a single copy of the shared segment simultaneously.

-o *file* Write output to *file* (default, **l.out**.)

-R *value*
Relocation base option. By default, **ld** links executeable files to run at the *user-base* for the computer. In almost all cases, the *user-base* is zero. If the **-R** option is used, **ld** will link the executeable to run at *value* instead of at zero. *value* can be set to any C-style constant, or to a symbol name that **ld** can find in the object files and archives being linked; remember that a C-accessible symbol *must* end with an underscore character '_'. This option is used primarily to produce output files that can be burned into ROM. These programs must make their own provisions for relocating initialized data and other tasks.

-r Retain relocation information in the output, and issue no diagnostic message for undefined symbols. By default **ld** discards relocation information from the output if there are no undefined symbols.

-s Strip the symbol table from the output. The same effect may be obtained by using the command **strip**. The **-s** and **-r** options are mutually exclusive.

-u *symbol*
Add *symbol* to the symbol table as a global reference, usually to force the linking of a particular library module.

-X Discard local compiler-generated symbols of the form '**L...**'.

-x Discard all local symbols.

Files

l.out — Default output
/coherent for **-k** option
/lib/lib*.a — Libraries
/usr/lib/lib*.a — More libraries

See Also

ar, ar.h, as, cc, commands, l.out.h, strip

Notes

By default, COHERENT allocates two kilobytes of stack to a process. This is sufficient for most processes. To change the amount of stack used by a given executable program, use the command **fixstack**. See its Lexicon entry for details. If you are linking a program by hand (that is, running **ld** independently from the **cc** command), be sure to include the appropriate run-time start-up routine with the **ld** command line; otherwise, the program will not link correctly.

ldexp() — General Function (libc)

Combine fraction and exponent
double ldexp(*f***, *e*) double** *f***; int** *e***;**

ldexp combines the fraction *f* with the binary exponent *e* to return a floating-point value *real* that satisfies the equation *real=m*2^e*.

See Also

atof(), ceil(), fabs(), floor(), frexp(), general functions, modf()

ldiv() — General Function (libc)

Perform long integer division
#include <stdlib.h>
ldiv_t ldiv(*numerator***, *denominator*)**
long *numerator*, *denominator*;

ldiv divides *numerator* by *denominator*. It returns a structure of the type **ldiv_t**, which is structured as follows:

```
typedef struct {
        long quot;
        long rem;
} ldiv_t;
```

ldiv writes the quotient into **quot** and the remainder into **rem**.

The sign of the quotient is positive if the signs of the arguments are the same; it is negative if the signs of the arguments differ. The sign of the remainder is the same as the sign of the numerator.

If the remainder is non-zero, the magnitude of the quotient is the largest integer less than the magnitude of the algebraic quotient. This is not guaranteed by the operators / and %, which merely do what the machine implements for divide.

See Also

div, general functions

Notes

The ANSI Standard includes this function to permit a useful feature found in most versions of FORTRAN, where the sign of the remainder will be the same as the sign of the numerator. Also, on most machines, division produces a remainder. This allows a quotient and remainder to be returned from one machine-divide operation.

If the result of division cannot be represented (e.g., because *denominator* is set to zero), the behavior of **ldiv** is undefined. *Caveat utilitor.*

let — Command

Evaluate an expression
let [*expression*]

The command **let** is built into the Korn shell **ksh**. It evalutes *expression*; it returns zero if *expression* evaluates to non-zero status, and non-zero if it evaluates to zero status.

See Also

commands, ksh

lex — Command

Lexical analyzer generator
lex [-t][-v][*file*]
cc lex.yy.c -ll

Many programs, e.g., compilers, process highly structured input according to rules. Two of the most complicated parts of such programs are *lexical analysis* and *parsing* (also called *syntax analysis*). The COHERENT system includes two powerful tools called **lex** and **yacc** to help you construct these parts of a program. **lex** converts a set of lexical rules into a lexical analyzer, and **yacc** converts a set of parsing rules into a parser.

The output of **lex** may be used directly, or may be used by a parser generated by **yacc**.

lex reads a specification from the given *file* (or from the standard input if none), and generates a C function called **yylex()**. **lex** writes the generated function in the file **lex.yy.c**, or on standard output if you use the **-t** option. The **-v** option prints some statistics about the generated tables.

The tutorial on **lex** that appear in this manual describes **lex** in detail. In brief, the generated function **yylex()** matches portions of its input to one pattern (sometimes called a regular expression) from a set of rules, or *context*, and executes associated C commands. Unmatched portions of the input are copied to the output stream. **yylex()** returns EOF when input has been exhausted.

lex uses the following macros that you may replace with the preprocessor directive **#undef** if you wish: **input()** (read the standard input stream), and **output(*c*)** (write the character *c* to the standard output stream). You may also replace the following functions if you wish: **main()** (main function), **error(...)** (print error messages; takes same arguments as **printf**), and **yywrap()** (handle events at the end of a file). If an action is desired on end of file, such as arranging for more input, **yywrap()** should perform it, returning zero to keep going.

A full **lex** specification has the following format:

- Macro definitions, of the form:
```
name pattern
```
- Start condition declarations:
```
%S    NAME ...
```
- Context declarations:
```
%C    NAME ...
```
- Code to be included in the header section:
```
%{
anything
%}
<tab or space> anything
```

- Rules section delimiter (must always be present):

```
%%
```

- Code to appear at the start of **yylex()**:

```
<tab or space> anything
```

- Rules for initial context, in any of the forms:

```
rule        action;
rule        | (means use next action)
rule        {
<tab or space>   action;
<tab or space>   }
```

- For each additional context:

```
%C    NAME
...rules for this context...
```

- End of rules section delimiter:

```
%%
```

- Code to be copied verbatim, such as user provided **input()**, **output()**, **yywrap()**, or other.

lex matches the longest string possible; if two rules match the same length string, the rule specified first takes precedence. **lex** puts the matched string, or *token*, in the **char** array **yytext[]**, and sets the variable **yyleng** to its length.

Actions may use the following:

ECHO	Output the token
REJECT	Perform action for lower precedence match
BEGIN *NAME*	Set start condition to *NAME*
BEGIN 0	Clear start condition
yyswitch(*NAME*)	Switch to context *NAME*, return current
yyswitch(0)	Switch to initial context
yynext()	Steal next character from input
yyback(*c*)	Put character *c* back into input
yyless(*n*)	Reduce token length to *n*, put rest back
yymore()	Append next token to this one
yylook()	Returns number of chars in input buffer

lex rules are contiguous strings of the form

[<*NAME*,...>][^] *token* [/*lookahead*][$]

where brackets '[ ]' indicate optional items.

<*NAME*,...>	Match only under given start conditions
^	Match the beginning of a line
$	Match the end of a line
token	Pattern that a given token is to match
/*lookahead*	Pattern that given trailing text is to match

Pattern elements:

a	The character **a**
\a	The character **a**, even if special
.	Any character except newline
[abx-z]	Any of **a**, **b**, or **x** through **z**
[^abx-z]	Any except **a**, **b**, or **x** through **z**
abc	The string **abc**, even if any are special
{*name*}	The macro definition *name*
(*exp*)	The pattern *exp* (grouping operator)

Optional operators on elements:

*e***?**	Zero or one occurrence of *e*
*e******	Zero or more consecutive *e*s
*e***+**	One or more consecutive *e*s
e{*n*}	*n* (a decimal number) consecutive *e*s
e{*m*,*n*}	*m* through *n* consecutive *e*s

Patterns may be of the form:

e1e2 Matches the sequence *e1 e2*
e1 | *e2* Matches either *e1* or *e2*

lex recognizes the standard C escapes: **\n**, **\t**, **\r**, **\b**, **\f**, and *ooo* (octal representation). The special characters

```
\ ( ) < > { } % * + ? [ - ] ^ / $ . |
```

must be prefixed with \ or enclosed within quotation marks (excepting " and \) to be normal. Within classes, only the characters . ^ - \ and] are special.

Files

/usr/lib/libl.a
/usr/src/libl/* — library source code

See Also

commands, yacc
Introduction to lex, the Lexical Analyzer

Lexicon — Introduction

The Mark Williams Lexicon is a new approach to documentation of computer software. The Lexicon is designed to improve documentation and eliminate some limitations found in more conventional documentation.

How to Use the Lexicon

The Lexicon consists of one large document that contains entries for every aspect of COHERENT. You will not have to search through a number of different manuals to find the entry you are looking for.

Every entry in the Lexicon has the same structure. The first line gives the name of the topic being discussed, followed by its type (e.g., **Command**).

The next lines briefly describe the item, then give the item's usage, where applicable. These are followed by a brief discussion of the item, and an example.

Cross-references follow. These can be to other entries or to other texts. Diagnostics and notes, where applicable, conclude each entry.

Internally, the Lexicon has a tree structure. The "root" entry is the present entry, for **Lexicon**. Below this entry comes the set of *Overview* entries. Each Overview entry introduces a group of entries; for example, the Overview entry for **string** introduces all of the string functions and macros, lists them, and gives a lengthy example of how to use them.

Each entry cross-references other entries. These cross-references point up the documentation tree, toward an overview article and, ultimately, to the entry for **Lexicon** itself. They also point down the tree to subordinate entries, and across to entries on related subjects. For example, the entry for **getchar** cross-references **STDIO**, which is its Overview article, plus **putchar** and **getc**, which are related entries of interest to the user. The Lexicon is designed so that you can trace from any one entry to any other, simply by following the chain of cross-references up and down the documentation tree.

Use the Lexicon

If, while reading an entry, you encounter a technical term that you do not understand, look it up in the Lexicon. You should find an entry for it. For example, if a function is said to return a data type **float** and you do not know exactly what a **float** is, look it up. You will find it described in full. In this way, you should increase your understanding of COHERENT, and make your programming easier and more productive.

Overview Articles

The Lexicon includes the following overview articles. Look at the appropriate overview article for information on the subject in which you are interested. The overview article will give you an overview of the topic, and tell you which Lexicon articles you should read to find detailed information.

C language
: This article summarizes COHERENT's implementation of the C lanaguage. It introduces subordinate articles, such as those that describe each C keyword.

commands
: This article briefly summarizes each COHERENT system command.

definitions
: The Lexicon includes a number of articles that define technical terms that are used through it. This overview article lists the definition articles included in the Lexicon.

device drivers
: This article introduces COHERENT's suite of device drivers, and points to subordinate articles that describe each driver in details.

environmental variables
: This article lists the commonly used environmental variables that are described in the Lexicon.

file formats
: The COHERENT system has a number of special files that contain information presented in a special format. Some files are meant to be read mechanically, such as executable files or relocatable objects; others you can edit to change the behavior of one or another COHERENT system. This overview article introduces the subordinate articles that describe the formats of these special files.

system maintenance
: Certain files and commands are used only to help you maintain your COHERENT system and help it run smoothly. This article introduces the subordinate articles that describe the COHERENT system's tools for to help you perform system

maintenance.

technical information
Finally, the Lexicon contains a set of articles that do not easily fit into any other category. These give broad technical information, both to help you decypher other articles within the Lexicon, and to provide you with a "cookbook" with which you can solve common problems. The article names should be self-explanatory, e.g., **terminal**, **printer**, and **RS-232**. If you're trying to tackle a new problem and don't have a clue as to where to begin, check this overview article first. You may well find that it lists a subordinate article that is helpful.

libraries — Overview

A **library** is an archive file of commonly used functions that have been compiled, tested, and stored for inclusion in a program at link time.

The COHERENT system stores its libraries in two directories, **/usr/lib** and **/lib**. The following libraries are kept in **/usr/lib**:

libcurses.a	**curses** library
libl.a	**lex** library
libmp.a	Multi-precision arithmetic library
libterm.a	**termcap** library
liby.a	**yacc** library
lib.b	**bc**'s function library (in **bc** source)

The following libraries are kept in **/lib**:

libc.a	General functions and system calls
libm.a	Mathematics routines

Library Functions

The following overview articles introduce the library functions included with the COHERENT system:

COHERENT system calls
ctype macros
curses
general functions
mathematics library
multiple-precision mathematics
STDIO
string functions
terminal-independent operations
time
variable arguments

See Also

ar, **C language**

limits.h — Header File

Define numerical limits
#include <limits.h>

The header file **limits.h** defines macros that set the numerical limits for the translation environment. It is described in sections 2.2.4.2 and 4.1.4 of the ANSI Standard.

The following table gives the macros defined in **limits.h**. Each value given is the macro's minimum maximum: a conforming implementation of C must meet these limits, and may exceed them.

CHAR_BIT
Number of bits in a **char**. This must be at least eight.

CHAR_MAX
Largest value representable in an object of type **char**. If the implementation defines a **char** to be signed, then it is equal to the value of the macro **SCHAR_MAX**; otherwise, it is equal to the value of the macro **UCHAR_MAX**.

CHAR_MIN
Smallest value representable in an object of type **char**. If the implementation defines a **char** to be signed, then it is equal to the value of the macro **SCHAR_MIN**; otherwise, it is zero.

INT_MAX
Largest value representable in an object of type **int**; it must be at least 32,767 (0x7FFF).

INT_MIN
Smallest value representable in an object of type **int**; it must be at most -32,767 (0x8000).

LONG_MAX
Largest value representable in an object of type **long int**; it must be at least 2,147,483,647 (0x7FFFFFFFL).

LONG_MIN
Smallest value representable in an object of type **long int**; it must be at most -2,147,483,647 (0x80000000L).

MB_LEN_MAX
Largest number of bytes in any multibyte character, for any locale; it must be at least one.

SCHAR_MAX
Largest value representable in an object of type **signed char**; it must be at least 127.

SCHAR_MIN
Smallest value representable in an object of type **signed char**; it must be at most -127.

SHRT_MAX
Largest value representable in an object of type **short int**; it must be at least 32,767 ((**short**)0x7FFF).

SHRT_MIN
Smallest value representable in an object of type **short int**; it must be at most -32,767 ((**short**)0x8000).

UCHAR_MAX
Largest value representable in an object of type **unsigned char**; it must be at least 255.

UINT_MAX
Largest value representable in an object of type **unsigned int**; it must be at least 65,535 ((**unsigned int**)0xFFFF).

ULONG_MAX
Largest value representable in an object of type **unsigned long int**; it must be at least 4,294,967,295 ((**unsigned long**)0xFFFFFFFFL).

USHRT_MAX
Largest value representable in an object of type **unsigned short int**; it must be at least 65,535 ((**unsigned short**)0xFFFF).

See Also
header files

lines — Command
Highly amusing board game
/usr/games/lines

lines is an interactive COHERENT version of a two-player board game by Claude Soucie called *Lines of Action*. The screen displays the game board with "X" and "O" characters marking the positions of the pieces. To see the rules of the game, type "r" and then press **<Enter>**. To see the available interactive commands, type "h" and press **<Enter>**.

Two players can use **lines** to keep track of a game between them by moving with the "M" command. Alternatively, one player can play against the computer by moving with the "m" command. The program uses a tree-search technique to consider possible moves; the player can vary the speed of the program's replies with commands that change the tree search width and depth.

For a more detailed description of *Lines of Action*, see *A Gamut of Games* by Sid Sackson (New York, Random House, 1969).

See Also
commands

link() — COHERENT System Call
Create a link
link(*old, new*)
char **old*, **new*;

A *link* to a file is another name for the file. All attributes of the file appear identical among all links.

link creates a link called *new* to an existing file named *old*.

For administrative reasons, it is an error for users other than the superuser to create a link to a directory. Such links can make the file system no longer tree structured unless carefully controlled, posing problems for commands such as **find**.

Example
This example, called **lock.c**, demonstrates how **link** can be used to perform intertask locking. With this technique, a program can start a process in the background and stop any other user from starting the identical process.

```
main()
{
	if(link("lock.c", "lockfile") == -1) {
		printf("Cannot link\n");
		exit(1);
	}

	sleep(50); /* do nothing for 50 seconds */
	unlink("lockfile");
	printf("done\n");
	exit(0);
}
```

See Also

COHERENT system calls, find, ln, unlink()

Diagnostics

link returns zero when successful. It returns -1 on errors, e.g., *old* does not exist, *new* already exists, attempt to link across file systems, or no permission to create *new* in the target directory.

Notes

Because each mounted file system is a self-contained entity, links between different mounted file systems fail.

linker-defined symbols — Overview

The following symbols are set by the linker **ld** when it links a program together:

edata	Location after shared and private data
end	Location after uninitialized data segment
etext	Location after text segments

These symbols may be read from the program's symbol table, e.g., for debugging.

See Also

C language, ld

ln — Command

Create a link to a file
ln [**-f**] *oldfile* [*newfile*]
ln [**-f**] *oldfile* ... *directory*

A *link* lets you give you a file more than one name. The file's contents or attributes may be changed via either name.

In its first form, **ln** links the name *newfile* to the file that is already named *oldfile*, provided that *newfile* does not already exist. If *newfile* is omitted, a link is created in the current directory with the same file name as *oldfile*, with leading directory information removed.

In the second form, **ln** links *oldfile* with an identical name in another *directory*. In effect, one file will "live" in two directories.

If *newfile* already exists, **-f** forces **ln** to unlink it and assign its name to *oldfile*.

Links to directories or across file systems are impossible.

See Also

commands, cp, ls, mv, rm

localtime() — Time Function (libc)

Convert system time to calendar structure
#include <time.h>
#include <sys/types.h>
tm *localtime(*timep***) time_t *****timep;**

localtime converts the COHERENT internal time into the form described in the structure **tm**.

timep points to the system time. It is of type **time_t**, which is defined in the header file **types.h**.

localtime returns a pointer to the structure **tm**, which is also defined in **time.h**. The function **asctime** turns **tm** into an ASCII string.

Unlike its cousin **gmtime**, **localtime** returns the local time, including conversion to daylight saving time, if applicable. The daylight saving time flag indicates whether daylight saving time is now in effect, *not* whether it is in effect during some part of the year. Note, too, that the time zone is set by **localtime** every time the value returned by

```
getenv("TIMEZONE")
```

changes. See the Lexicon entry for **TIMEZONE** for more information on how COHERENT handles time zone settings.

Example

The following example recreates the function **asctime**. It builds a string somewhat different from that returned by **asctime** to demonstrate how to manipulate the **tm** structure.

```
#include <time.h>
#include <sys/types.h>

char *month[] = {
      "January", "February", "March", "April",
      "May", "June", "July", "August", "September",
      "October", "November", "December"
};

char *weekday[] = {
      "Sunday", "Monday", "Tuesday", "Wednesday",
      "Thursday", "Friday", "Saturday"
};

main()
{
      char buf[20];
      time_t tnum;
      struct tm *ts;
      int hour = 0;
```

```
        time(&tnum);     /* get time from system */

        /* convert time to tm struct */
        ts=localtime(&tnum);

        if (ts->tm_hour == 0)
             sprintf(buf,"12:%02d:%02d A.M.",
                  ts->tm_min, ts->tm_sec);

        else
             if(ts->tm_hour>=12) {
                  hour=ts->tm_hour-12;
                  if (hour==0)
                       hour=12;
                  sprintf(buf,"%02d:%02d:%02d P.M.",
                       hour, ts->tm_min,ts->tm_sec);

             } else
                  sprintf(buf,"%02d:%02d:%02d A.M.", ts->tm_hour,
                       ts->tm_min,ts->tm_sec);

        printf("\n%s %d %s 19%d %s\n",
             weekday[ts->tm_wday], ts->tm_mday,
             month[ts->tm_mon], ts->tm_year, buf);

        printf("Today is the %d day of 19%d\n",
             ts->tm_yday, ts->tm_year);

        printf("Daylight Saving Time %s in effect\n",
             ts->tm_isdst ? "is" : "is not");
}
```

See Also

gmtime(), time, TIMEZONE

Notes

localtime returns a pointer to a statically allocated data area that is overwritten by successive calls.

log() — Mathematics Function (libm)

Compute natural logarithm

#include <math.h>
double log(*z*) double *z*;

log returns the natural (base e) logarithm of its argument *z*.

Example

For an example of this function, see the entry for **exp**.

See Also

log10(), mathematics library

Diagnostics

A domain error in **log** (z is less than or equal to zero) sets **errno** to **EDOM** and returns zero.

log10() — Mathematics Function (libm)

Compute common logarithm
#include <math.h>
double log10(z) **double** z;

log10 returns the common (base 10) logarithm of its argument z.

Example

For an example of this function, see the entry for **exp**.

See Also

log(), mathematics library

Diagnostics

A domain error in **log10** (z is less than or equal to zero) sets **errno** to **EDOM** and returns zero.

login — Command

Log in or change user name
login [*username*]

The COHERENT system normally invokes **login** as part of the log in sequence on an unused terminal. The user may also invoke **login** directly from the shell **sh**, usually to change to a different user name. If *username* is not present, **login** prompts the user. If the account has a password, **login** asks for it.

If the user logs in successfully, **login** then reads the file **/etc/motd** (which holds the "message of the day") and prints its contents on the screen, then notifies the user if mail is waiting to be read. It then sets the working directory to the user's base directory and sets the user id and group id, transfers ownership of the tty to the user, and updates the login accounting file. Finally, if a program is specified in **/etc/passwd**, **login** reads **/etc/profile** for lines beginning "export", inserts the remainder of the line into the environment, then executes the specified program. If the program field is blank, **login** executes **sh**, which executes the contents of **$HOME/.profile** if it is present.

From the shell, a user may log in by typing **login** or by typing an end of file (normally **<ctrl-D>**) to terminate the previous shell.

When the superuser **root** logs in, **login** sets **HOME** to '/' and reads the file **/.profile** should one exist.

Files

/etc/logmsg — Login message (default, "Coherent login:")
/etc/passwd — User information
/etc/profile — System profile
/etc/motd — Message of the day
/etc/utmp — Users currently using system
/usr/adm/wtmp — Login accounting history
/usr/adm/failed — Log of failed login attempts
$HOME/.profile — User profile

See Also

ac, commands, getty, sh, su, utmp.h

Notes

Under the Korn shell, **login** is an alias for the expression **exec login**.

logmsg — System Maintenance

Hold COHERENT Login Message
/etc/logmsg

The file **/etc/logmsg** holds the message that COHERENT displays to prompt the user to log in. The superuser **bin** can use **ed** or MicroEMACS to change the message to whatever she prefers.

See Also

system maintenance

Notes

The default message consists of the bell character **<ctrl-G>** followed by the text **Coherent login:**. If the bell annoys you, simply delete the **<ctrl-G>** from **/etc/logmsg**.

long — C Keyword

Data type

A **long** is a numeric data type. By definition, a **long** is the largest integer data type. It cannot be smaller than an **int**, although on some machines an **int** and a **long** will be the same size. Under COHERENT, **sizeof long** equals two machine words, or four **chars** (31 data bits plus a sign bit).

See Also

C keywords, data formats, int

longjmp() — General Function (libc)

Return from a non-local goto
#include <setjmp.h>
int longjmp(*env*, *rval*) **jmp_buf** *env*; **int** *rval*;

The function call is the only mechanism that C provides to transfer control between functions. This mechanism is inadequate for some purposes, such as handling unexpected errors or interrupts at lower levels of a program. To answer this need, **longjmp** provides a non-local *goto*.

longjmp restores an environment that had been saved by a previous call to the function **setjmp**. It returns the value *rval* to the caller of **setjmp**, just as if the **setjmp** call had just returned. Note that **longjmp** must not restore the environment of a routine that has already returned. The type declaration for **jmp_buf** is in the header file **setjmp.h**. The environment saved includes the program counter, stack pointer, and stack frame. These routines do not restore register variables in the environment returned.

Example

For an example of this function, see the entry for **longjmp**.

See Also

general functions, setjmp()

Notes

Programmers should note that many user-level routines cannot be interrupted and reentered safely. For that reason, improper use of **longjmp** and **setjmp** can result in the creation of mysterious and irreproducible bugs. Do not attempt to use **longjmp** within an exception handler.

look — Command

Find matching lines in a sorted file
look [-df] *string* [*file*]

The command **look** scans the sorted *file* and prints each line that begins with *string*.

The following options specify the order of the search:

-d Use dictionary order: the only characters tested are alphanumerics and blanks.

-f Convert all alphabetic characters to upper case.

If no *file* is specified, **look** uses **/usr/dict/words** with the **-df** option.

Files

/usr/dict/words — File of words (sorted with **sort -df**).

See Also

commands, sort

Notes

Because the file **/usr/dict/words** is quite large, it might not be included with COHERENT systems for machines with limited disk space. As a result, the command might not work as expected on all systems.

lp — Device Driver

Line printer driver

Files **/dev/lp*** access the line-printer's device drivers for IBM AT COHERENT. The drivers are assigned major device number 3.

The COHERENT system supports three printers, in both cooked and raw modes. The following gives the device name, minor device, and I/O port:

/dev/lpt1	0	0x3BC	(/etc/mknod /dev/lpt1 c 3 0)
/dev/lpt2	1	0x378	(/etc/mknod /dev/lpt2 c 3 1)
/dev/lpt3	2	0x278	(/etc/mknod /dev/lpt3 c 3 2)
/dev/rlpt1	128	0x3BC	(/etc/mknod /dev/rlpt1 c 3 128)
/dev/rlpt2	129	0x378	(/etc/mknod /dev/rlpt2 c 3 129)
/dev/rlpt3	130	0x278	(/etc/mknod /dev/rlpt3 c 3 130)

"Cooked" processing processes the special characters BS (backspace), HT (horizontal tab), LF (line feed), FF (form feed), and CR (carriage return) appropriately; raw processing simply passes them on to the printer.

The driver uses a hybrid busy-wait/timeout discipline to support printers efficiently that have varying buffer sizes in a multi-tasking environment.

The kernel variable **LPWAIT_** sets the time for which the processor waits for the printer to accept the next character. If the printer is not ready within the **LPWAIT_** period, the processor then resumes normal processing for the number of ticks set by by the kernel variable **LPTIME_**. Thus, setting **LPWAIT_** to an extremely number (e.g., 1,000) and

LPTIME_ to a very small number (e.g., one) results in a fast printer, but leaves very few CPU cycles available for anything else. Conversely, setting **LPWAIT_** to a small number (e.g., 50) and **LPTIME_** to a large number (e.g., five) result in efficient multi-tasking but also results in a slow printer unless the printer itself contains a buffer (as is normal with all but the least expensive printers). By default, **LPWAIT_** is set to 400 and **LPTIME_** to four.

We recommend that you set **LPWAIT_** to no less than 50 and no more than 1,000, and **LPTIME_** to no less than one. To change the values of **LPWAIT_** to 500 and **LPTIME_** to one, use the following command:

```
/conf/patch -k /coherent LPWAIT_=500 LPTIME_=1
```

The kernel variable **LPTEST_** determines whether the device driver checks to see if the printer is in an "on-line" condition before it uses the device. If your printer does not support this signal, you must set **LPTEST_** to zero.

Files

/dev/lp* — "Cooked" printer interfaces
/dev/rlp* — Raw printer interfaces

See Also

ascii, db, device drivers, epson, lpr

lpd — System Maintenance

Line printer spooler daemon
/usr/lib/lpd

lpd is a daemon program that runs in the background and prints listings queued by the command **lpr**. It is run automatically by **lpr.** If there is no printing to do, or if another daemon is already running (indicated by the file **dpid**), **lpd** exits immediately. Otherwise, it searches the spool directory for control files of listings to print. These control files contain the names of files to print, the user name, banners, and files to be removed upon completion.

lpd does not print listings in any particular order. Priority is not given to any file, either by size or by requester.

The command **lpskip** command terminates or restarts the current line printer listing.

Files

/dev/lp — Printer
/usr/spool/lpd — Spool directory
/usr/spool/lpd/cf* — Control files
/usr/spool/lpd/df* — Data files
/usr/spool/lpd/dpid — Lock and process id

See Also

init, lpr, lpskip, system maintenance

lpioctl.h — Header File

Definitions for line-printer I/O control
#include <sys/lpioctl.h>

lpioctl.h defines constants used by routines that control I/O on the line printer.

See Also

header files

lpr — Command

Send to line printer spooler
lpr [-cmnr] [-b *banner*] [*file* ...]

lpr lets a user print each specified *file* on the line printer, without conflicting with printing by other users. If no *file* is specified, **lpr** prints the standard input on the line printer.

lpr recognizes the following options:

-B Suppress printing of a banner.

-b *banner*
Print *banner* at the beginning of the file. The default banner is the user's login name.

-c Copy the files (allowing changes to be made before the printing completes).

-m Send a message when the printing completes.

-n Do not send a message (default).

-r Remove the files when they have been spooled.

The command **lpskip** aborts or restarts the current listing.

Files

/dev/lp — Line printer
/usr/lib/lpd — Line printer daemon
/usr/spool/lpd — Spool directory
/usr/spool/lpd/dpid — Daemon lockfile

See Also

commands, lpd, lpskip, pr, printer

lpskip — Command

Terminate/restart current line printer listing
lpskip [-r]

The command **lpskip** aborts or restarts the printing of a file. By default, **lpskip** aborts the current listing and prints a diagnostic message. When invoked with the **-r** option, it restarts the current listing. This is useful when a printing is spoiled due to, say, a paper jam.

lpskip works only with files that have been spooled to the line printer via the command **lpr**.

Files

/usr/lib/lpd — Line printer daemon
/usr/spool/lpd — Spool directory
/usr/spool/lpd/dpid Daemon lockfile

See Also

commands, lpd, pr

filename # - generated by mail on DC

ls — Command

List directory's contents

ls [-abCcdFfgilmnopqRrstux] [*file* ...]

The command **ls** prints information about each *file*. Normally, **ls** sorts its output by file name and prints only the name of each *file*. If a directory name is given as an argument, **ls** sorts and lists its contents, not including '.' and '..'. If no *file* is named, **ls** lists the contents of the current directory.

The following options control how **ls** sorts and displays its output:

-a Print all directory entries, including '.', '..', any hidden files, and volume ID's.

-b Print non-graphic characters in octal.

-C Print the output in multi-column format, sorted down the columns.

-c Print the time the files' attributes were last changed.

-d Treat directories as if they were files.

-F Print a '/' after the name of each directory, and print an '*' after each executable file.

-f Force each argument to be treated as a directory. This disables the **-lrst** options and sorting, and enables the **-a** option.

-g Same as **-l** except that the user id is not displayed.

-i Print the i-number of each file.

-l Print information in long format. The fields give mode bits, link count, owner uid, owner gid, size in bytes, date, and file name. For special files, major and minor device numbers replace the size field.

-m "Stream" the output horizontally across the screen, with each file name separated by a comma.

-n Same as **-l**, except the group identifiers and user identifiers are numbers rather than names.

-o Same as **-l**, except that the group id is not printed.

-p Print a '/' after the each directory name.

-q Print non-graphics characters as '?'.

-r Reverse the sense of the sort.

-R Recursively print directories.

-s Print the size in blocks of each file.

-t Sort by time, newest first.

-u Sort by the *access* time.

-x Print multicolumn output, sorted across the columns. This resembles the output of the command **lc**.

The date **ls** prints with the **-l** and **-t** options is the *modification* time, unless the **-c** or **-u** option is used as well.

The mode field in the long list format consists of ten characters. The first character will be one of the following:

-	Regular file
b	Block special file
c	Character special file
d	Directory
p	Pipe
x	Bad entry (remove it immediately!)

The remaining nine characters are permission bits, in three sets of three characters each. The first set pertains to the owner of the file, the second to users from the owner's group, and the third to users from other groups. Each set may contain 3 characters from the following.

r	The file can be read
s	Set effective user ID or group ID on execution
t	Shared text is sticky
w	The file can be written
x	The file is executable
-	No permission is given

See Also

chmod, commands, lc, stat

lseek() — COHERENT System Call (libc)

Set read/write position
long lseek(*fd*, *where*, *how*)
int *fd*, *how*; **long** *where*;

lseek changes the *seek position*, or the point within a file where the next read or write operation is performed. *fd* is the file's file descriptor, which is returned by **open**.

where and *how* describe the new seek position. *where* gives the number of bytes that you wish to move the seek position; it is measured from the beginning of the file if *how* is zero, from the current seek position if *how* is one, or from the end of the file if *how* is two. A successful call to **lseek** returns the new seek position. For example,

```
position = lseek(filename, 100L, 0);
```

moves the seek position 100 bytes past the beginning of the file; whereas

```
position = lseek(filename, 0L, 1);
```

merely returns the current seek position, and does not change the seek position at all.

Sparse files may be created by seeking beyond the current size of the file and writing. The "hole" between the end of the file and where the write occurs is read as zero and will occupy no disk space. For example, if you **lseek** 10,000 bytes past the current end of file and write a string, the data will be written 10,000 bytes past the old end of file and all intervening matter will be considered part of the file.

lseek differs from its cousin **fseek** in that **lseek** is a system call and uses a file descriptor, whereas **fseek** is a C function and uses a **FILE** pointer.

See Also

COHERENT system calls, STDIO

Diagnostics

lseek returns **-1L** on an error, such as seeking to a negative position. If no error occurs, it returns the new seek position.

Notes

lseek is permitted on character-special files, but drivers do not generally implement it. As a result, seeking a terminal will not generate an error but will have no discernible effect.

ltol3() — General Function (libc)

Convert long integer to file system block number

ltol3(*l3p, lp, n*)
char **l3p*;
long **lp*;
unsigned *n*;

To conserve space inside i-nodes in COHERENT file systems, the system stores block addresses in three bytes. Programs that reference or maintain file systems use the functions **l3tol** and **ltol3** to convert between the three byte representation and **long** numbers.

ltol3 converts *n* **long** integers at address *lp* to the more compact form at address *l3p*.

See Also

canon.h, general functions, l3tol()

lvalue — Definition

An **lvalue** is an expression that designates a region of storage. The name comes from the assignment expression **e1=e2;**, in which the left operand must be an lvalue.

An identifier has both an *lvalue* (its address) and an *rvalue* (its contents). Some C operators require lvalue operands; for example, the left operand of an assignment statement must be an lvalue. Some operators give lvalue results; for example, if *e* is a pointer expression, **e* is an lvalue that designates the object to which *e* points.

A *variable* can be used as an lvalue, whereas a constant cannot. For example, you cannot say

```
6 = (foo+bar);
```

A pointer is a variable, and can be manipulated within limits. An array name, however, is a constant and cannot be altered legally. Thus, the code

```
int foo[10];
int *bar;
foo = bar;
```

will generate an error message when you attempt to compile it, whereas

```
int foo[10];
int *bar;
bar = foo;
```

will not.

The following example shows the use of both an lvalue and a rvalue:

```
int i, *ip;

ip = &i;        /* ip is an lvalue, i and &i are rvalues */
i = 3;          /* i is an lvalue, 3 is an rvalue */
*ip = 4;        /* *ip is an lvalue, 4 is an rvalue */
```

See Also

definitions, rvalue

M

m4 — Command

Macro processor
m4 [file ...]

The command **m4** processes macros. It allows you to define strings for which **m4** is to search, and strings to replace them; **m4** then opens *file*, reads its contents, replaces each macro with its specified replacement string, and writes the results into the standard output stream.

m4 can also perform file manipulation, conditional decision making, substring selection, and arithmetic. The *Introduction to the m4 Macro Processor* describes **m4** in detail.

The *files* are read in the order given; if no *file* is named, then **m4** reads the standard input stream. The file name '-' indicates the standard input.

m4 copies input to output until it finds a potential *macro*. A macro is a string of alphanumerics (letters, digits, or underscores) that begins with a non-digit character and is surrounded by non-alphanumerics. If **m4** does not recognize the *macro*, it simply copies it to the output and continues processing. If **m4** recognizes the *macro* and the next character is a left parenthesis '(', an *argument set* follows:

```
macro(arg1,..., argn)
```

The arguments are collected by processing them in the same manner as other text (thus, an arguments may itself be another macro), and resulting output text is diverted into storage. **m4** stores up to nine arguments; any more will be processed but not saved. An argument set consists of strings of text separated by commas (commas inside quotation marks or parentheses do not terminate an argument), and must contain balanced parentheses that are free of quotation marks (i.e., that are *unquoted*). **m4** strips arguments of unquoted leading space (blanks, tabs, newline characters).

m4 then removes the *macro* and its optional argument set from the input stream, processes them, and replaces them in the input stream with the resulting value. The value becomes the next piece of text to be read.

Quotation marks, of the form ` ', inhibit the recognition of *macro*. **m4** strips off one level of quotation marks when it encounters them (quotation marks are nestable). Thus, '*macro*' is not processed, but is changed to *macro* and passed on.

m4 determines the *value* of a user-defined macro by taking the text that constitutes the macro's *definition* and replacing any occurrence within that text of '$*n*' (where *n* is '0' through '9') with the text of the *n*th argument. Argument 0 is the *macro* itself.

m4 recognizes the following predefined macros:

changequote[([*openquote*],[*closequote*])]
: Changes the quotation characters. Missing arguments default to ` for open or ' for close. Quotation characters will not nest if they are defined to be the same character. Value is null.

decr[(*number*)]
: Decrement *number* (default, 0) by one and returns resulting value.

define(*macro*,*definition*)
: Define or redefine *macro*. If a predefined macro is redefined, its original definition is irrecoverably lost. Value is null.

divert[(*n*)]
Redirects output to output stream *n* (default is 0). The standard output is 0, and 1 through 9 are maintained as temporary files. Any other *n* results in output being thrown away until the next **divert** macro. Value is null.

divnum
Value is current output stream number.

dnl Delete to newline: removes all characters from the input stream up to and including the next newline. Value is null.

dumpdef[(*macros*)]
Value is quoted definitions of all *macros* specified, or names and definitions of all defined macros if no arguments.

errprint(*text*)
Print *text* on standard error file. Value is null.

eval(*expression*)
Value is a number that is the value of evaluated *expression*. It recognizes, in order of decreasing precedence: parentheses, **, unary + -, * / %, binary + -, relations, and logicals. Arithmetic is performed in **longs**.

ifdef(*macro,defvalue,undefvalue*)
Return *defvalue* if *macro* is defined, and *undefvalue* if not.

ifelse(*arg1,arg2,arg3...*)
Compares *arg1* and *arg2*. If they are the same, returns *arg3*. If not, and *arg4* is the last argument, return *arg4*. Otherwise, the process repeats, comparing *arg4* and *arg5*, and so on. Like other **m4** macros, this takes a maximum of nine arguments.

include(*file*)
Value is the entire contents of the *file* argument. If *file* is not accessible, a fatal error results.

incr[(*number*)]
Increments given *number* (default, zero) by one and returns resulting value.

index(*text,pattern*)
Value is a number corresponding to position of *pattern* in *text*. If *pattern* does not occur in *text*, value is -1.

len(*text*)
Value is a number that corresponds to length of *text*.

maketemp(*filenameXXXXXX*)
Value is *filename* with last six characters, usually **XXXXXX**, replaced with current process id and a single letter. Same as system call **mktemp**.

sinclude(*file*)
Value is the entire contents of *file*. If *file* is not accessible, return null and continue processing.

substr(*text*[,*start*[,*count*]])
Value is a substring of *text*. *start* may be left-oriented (nonnegative) or right-oriented (negative). *count* specifies how many characters to the right (if positive) or to the left (if negative) to return. If absent, it is assumed to be large and of the same sign as *start*. If *start* is omitted, it is assumed to be zero if *count* is positive or omitted, or -1 if *count* is negative.

syscmd(*command*)
Pass *command* to the shell **sh** for execution. Value is null. Same as system call **system**.

translit(*text*,*characters*[,*replacements*])
Replaces *characters* in *text* with the corresponding characters from *replacements*. If the *replacements* is absent or too short, replace *characters* with a null character. Value is *text* with specified replacements.

undefine(*macro*)
Remove macro definition. Value is null. If a predefined macro is redefined, its original definition is irrecoverably lost.

undivert[(*stream*[,...])]
Dumps each specified *stream* into the current output stream. With no arguments, **undivert** dumps all output streams in numeric order. **m4** will not dump any output stream into itself. At the end of processing, **m4** automatically dumps all diverted text to standard output in numeric order. Value is null.

See Also

commands, mktemp, system
Introduction to the m4 Macro Processor

machine.h — Header File

Machine-dependent definitions
#include <sys/machine.h>

machine.h defines macros, constants, and structures that are specific to the machine upon which COHERENT is being run.

See Also

header files

macro — Definition

A **macro** is a body of text that is given a name. When the name is used in a program, it is replaced with the text to which it refers; this is called macro *expansion*. For example, **getchar** is a macro that consists of the function call **getc(stdin)**.

Because macros may employ an argument *n* times, any arguments that have side effects will have the side effect repeated *n* times as well, which may be undesirable.

See Also

#define, definitions, function, m4

madd() — Multiple-Precision Mathematics

Add multiple-precision integers
#include <mprec.h>
void madd(*a, b, c*)
mint **a, *b, *c;**

The COHERENT system includes a suite of routines that allow you to perform multiple-precision mathematics. **madd** sets the multiple-precision integer (or **mint**s) pointed to by *c* to the sum of the the **mint**s pointed to by *a* and *b*.

See Also

multiple-precision mathematics

mail — Overview

Electronic mail system

The COHERENT system includes a full-featured, UNIX-style mail system. It consists of a number of commands and files through which you can describe potential recipients of mail, either on your system or other systems, and send mail to them either directly or via UUCP. This article describes the design of the COHERENT mail system, and introduces the commands and files that compose it.

Structure of the COHERENT Mail System

The COHERENT mail system has three major components: the *user agent* (i.e., **mail**); the *routing agent* (the commands **smail** and **rmail**); and the delivery agents (the commands **lmail** and **uux**).

The user agent collects messages from the user and reads messages from a user's mailbox. It hands to the routing agent for delivery any messages it receives from the user.

The routing agents decode addresses and decide how to deliver a message. They are the only components of the mail system that must run **setuid** to assume the privilege of the superuser **root**.

The delivery agents move messages to their destination.

The local delivery agent, **lmail**, places messages into users' mailboxes. To discourage mail forging, **lmail** does not use **setuid**. It must be run by a privileged user (generally **root**) to write into all mail boxes. As a rule, **lmail** is invoked only by a routing agent.

The UUCP delivery agent, **uux**, queues messages for transmission to remote systems. It uses **setuid** to assume the identity of user **uucp** in order to write into the necessary spool directories. It has long been trivial to forge messages to remote systems with **uux**; keep this in mind if you plan to use electronic mail for any kind of authorization system.

smail looks up each *user* in file **/usr/lib/mail/aliases**. If it finds a match, it uses the matched name in place of *user*. If *user* is of the form

sys!*user*

or

sys! . . . !*user*

or

user@*sys*[.*domain*]

it is treated as a remote destination. **smail** then invokes command **uux** to pass the message to *sys*, whose responsibility it becomes to pass the message to *user*.

If **smail** finds no match in **/usr/lib/mail/aliases**, or **$HOME/.aliases**, however, it attempts to look up each *user* in file **/etc/passwd**, to see if this is a local user. If it does not find *user* in this file, **smail** mails an error message back to the sender. If, however, it does find *user* in this file, **smail** checks file **$HOME/.forward** for any possible forwarding instructions. If this file is absent, **smail** passes the message to **lmail**. **lmail** writes the message into the file **/usr/spool/mail/***user*. This file is called the user's "mailbox". Note that *user* owns this file, and can therefore permit or deny access to her mail by other users.

Note that before you can send mail, either locally or to a remote site, you must run the program **uuinstall** and use its 'S' option to set the name of your local site and domain. Your local system must, of course, also have permission to log into any remote site to which you wish to send mail. See the tutorial and Lexicon articles on UUCP for details on using UUCP to exchange mail and files with remote sites.

Files

$HOME/dead.letter — Message that **mail** could not send
$HOME/.aliases — Personal mail alias file for outgoing mail
$HOME/.forward — Forwarding instructions for inbound mail
/etc/domain — Name of your system's domain
/etc/passwd — User identities
/etc/uucpname — Name of your system
/tmp/mail* — Temporary and lock files
/usr/lib/mail/aliases — Aliases of users
/usr/lib/mail/fullnames — Short full name aliases of users
/usr/lib/mail/paths — Mail routing control file
/usr/spool/mail — Mailbox directory, filed by user name

See Also

aliases, commands, mail, mkfnames, msg, nptx, paths, uux

Notes

To mail a file to another user, use the shell's redirection operator '<'. For example, the command

```
mail stephen <bug.report
```

mails file **bug.report** to user **stephen**. The file will be prefixed with your address, and suffixed with your mail "signature", should you have one.

mail — Command

Computer mail
mail [-mpqrv] [-f*file***] [***user* ...**]**

mail allows you to exchange electronic mail with other COHERENT system users, either on your own system or on other systems via UUCP. Depending upon its form, this command can be used either to send mail to other users or to read the mail that other users have sent to you.

Sending Mail

If you name one or more *users*, **mail** assumes that you wish to send a mail message to each *user*. **mail** first prints the prompt

```
Subject:
```

on the screen, requesting that you give the message a title.

mail then reads what you type on the standard input. A message is terminated by **<ctrl-D>**, by a line that contains only the character '.', or by a line that contains only the character '?'. Ending with a question mark prompts **mail** to feed the message into an editor for further editing. The editor used is the one named in the environmental variable **EDITOR**. If this variable is not defined, **mail** uses **ed**.

If you have defined environmental variable **ASKCC** to **YES**, **mail** asks you, after a message is ended, for a list of users to whom you wish to send a copy of the message.

Finally, **mail** prepends the date and the sender's name, and sends the result to each *user* named either on the command line or on the carbon-copy list with the **rmail** command.

Each *user* who has received mail is greeted by the message "You have mail." when she logs in. **mail** normally changes the contents of the mailbox as the user works with them; however, **mail** has options that allow the contents of the mailbox to remain unchanged if the user desires.

Reading Mail

If no *user* is named on its command line, **mail** reads and displays the user's mail, message by message. If environmental variable **PAGER** is defined, **mail** will "pipe" each message through the command it names. For example, the **.profile** command line:

```
export PAGER="exec /bin/scat -1"
```

invokes **/bin/scat** for each mail message with the command-line argument **-1** (the digit one).

While reading mail, the user can use any of the following commands to save, delete, or send each message to another user interactively.

d Delete the current message and print the next message.

m [*user* ...]
Mail the current message to each *user* given (default: yourself).

p Print the current message again.

q Quit, and update mailbox file to reflect changes.

r Reverse the direction in which the mailbox is being scanned.

s [*file* ...]
Save the current mail message with the usual header in each *file* (default: **$HOME/mbox**).

t [*user* ...]
Send a message read from the standard input, terminated by an end-of-file character or by a line containing only '.' or '?', to each *user* (default: yourself).

w [*file* ...]
Write the current message without the usual header in each *file* (default: **$HOME/mbox**).

x Exit without updating the mailbox file.

<newline>
Print the next message.

- Print the previous message.

EOF Quit, updating mailbox; same as **q**.

? Print a summary of available commands.

!*command*
Pass *command* to the shell for execution.

The following command line options control the sending and reading of mail.

-f *file* Read mail from *file* instead of from the default, **/usr/spool/mail/***user*.

-m Send a message to the terminal of *user* if he is logged into the system when mail is sent.

-p Print all mail without interaction.

-q Quit without changing the mailbox if an interrupt character is typed. Normally, an interrupt character stops printing of the current message.

-r Reverse the order of printing messages. Normally, **mail** prints messages in the order in which they were received.

-v Verbose mode. Show the version number of the **mail** program, and display expanded aliases.

If you wish, you can create a signature file, **.sig.mail**, in your home directory. **mail** appends the contents of the signature file to the end of every mail message you send, as a signature. A signature can be your system's path name (for **uucp** messages), your telephone number, an amusing *bon mot*, or what you will.

Files

$HOME/dead.letter— Message that **mail** could not send
$HOME/mbox — Default saved mail
$HOME/.sig.mail — Signature file
/etc/domain — Name of your system's domain
/etc/uucpname — Name of your system
/tmp/mail* — Temporary and lock files
/usr/spool/mail — Mailbox directory, filed by user name

See Also

aliases, ASKCC, commands, EDITOR, mkfnames, msg, nptx, PAGER, paths, uux

Notes

Note that before you can send mail, either locally or to a remote site, you must run the program **uuinstall** and use its 'S' option to set the name of your local site and domain. Your local system must, of course, also have permission to log into any remote site to which you wish to send mail. See the tutorial and Lexicon articles on UUCP for details on using UUCP to exchange mail and files with remote sites.

main() — C Language

Introduce program's main function

A C program consists of a set of functions, one of which must be called **main**. This function is called from the runtime startup routine after the runtime environment has been initialized.

Programs can terminate in one of two ways. The easiest is simply to have the **main** routine **return**. Control returns to the runtime startup; it closes all open file streams and otherwise cleans up, and then returns control to the operating system, passing it the value returned by **main** as exit status.

In some situations (errors, for example), it may be necessary to stop a program, and you may not want to return to **main**. Here, you can use **exit**; it cleans up the debris left by the broken program and returns control directly to the operating system.

A second exit routine, called **_exit**, quickly returns control to the operating system without performing any cleanup. This routine should be used with care, because bypassing the cleanup will leave files open and buffers of data in memory.

Programs compiled by COHERENT return to the program that called them; if they return from **main** with a value or call **exit** with a value, that value is returned to their caller. Programs that invoke other programs through the **system** function check the returned value to see if these secondary programs terminated successfully.

See Also

_exit, argc, argv, C language, envp, exit

major number — Definition

Device numbering

A *major number* specifies the device driver associated with a given device name found in the directory **/dev**. COHERENT uses a device's the major number as an index into an internal table of device-driver pointers.

Every COHERENT device has a device number associated with it. This device number is of type **dev_t**, as defined in **<sys/types.h>**. The macro **major()** in **<sys/stat.h>** extracts the major number from a given device number.

See Also

device drivers, minor number, stat.h

make — Command

Program building discipline
make [*option* ...] [*argument* ...] [*target* ...]

make helps you build programs that consist of more than one file of source code.

Complex programs often consist of several *object modules*, each of which is the product of compiling a *source file*. A source file may refer to one or more **include** files, which can also be changed. Some programs may be generated from specifications given to program generators, such as **yacc**. Recompiling and relinking complicated programs can be difficult and tedious.

make regenerates programs automatically. It follows a specification of the structure of the program that you write into a file called **makefile**. **make** also checks the date and time that COHERENT has recorded for each source file and its corresponding object module; to avoid unnecessary recompilation, **make** will recompile a source file only if it has been altered since its object module was last compiled.

The Makefile

A **makefile** consists of three types of instructions: *macro definitions*, *dependency definitions*, and *commands*.

A macro definition simply defines a macro for use throughout the **makefile**; for example, the macro definition

```
FILES=file1.o file2.o file3.o
```

Note the use of the equal sign '='.

A dependency definition names the object modules used to build the target program, and source files used to build each object module . It consists of the *target name*, or name of the program to be created, followed by a colon ':' and the names of the object modules that build it. For example, the statement

```
example:  $(FILES)
```

uses the macro **FILES** to name the object modules used to build the program **example**. Likewise, the dependency definition

```
file1.o:  file1.c macros.h
```

defines the object module **file1.o** as consisting of the source file **file1.c** and the header file **macros.h**.

Finally, a command line details an action that **make** must perform to build the target program. Each command line must begin with a space or tab character. For example, the command line

```
        cc -o example $(FILES)
```

gives the **cc** command needed to build the program **example**. The **cc** command lists the *object modules* to be used, *not* the source files.

Note that if you prefix an action with a hyphen '-', **make** will ignore errors in the action. If the action is prefixed by '@', it tells **make** to be silent about the action — that is, do not echo the command to the standard output.

Finally, you can embed comments within a **makefile**. **make** recognizes any line that begins with a pound sign '#' as being a comment, and ignores it.

make searches for **makefile** first in directories named in the environmental variable **PATH**, and then in the current directory.

Dependencies

The **makefile** specifies which files depend upon other files, and how to recreate the dependent files. For example, if the target file **test** depends upon the object module **test.o**, the dependency is as follows:

```
test: test.o
        cc -o test test.o
```

make knows about common dependencies, e.g., that **.o** files depend upon **.c** files with the same base name. The target **.SUFFIXES** contains the suffixes that **make** recognizes.

make also has a set of rules to regenerate dependent files. For example, for a source file with suffix **.c** and a dependent file with the suffix **.o**, the target **.c.o** gives the regeneration rule:

```
.c.o:
        cc -c $<
```

The **-c** option to the **cc** commands tells **cc** not to link or erase the compiled object module. **$<** is a macro that **make** defines; it stands for the name of the file that causes the current action. The default suffixes and rules are kept in the files **/usr/lib/makemacros** and **/usr/lib/makeactions**.

Macros

To simplify the writing of complex dependencies, **make** provides a *macro* facility. To define a macro, write

NAME = string

string is terminated by the end-of-line character, so it can contain blanks. To refer to the value of the macro, use a dollar sign '$' followed by the macro name enclosed in parentheses:

```
$(NAME)
```

If the macro name is one character, parentheses are not necessary. **make** uses macros in the definition of default rules:

```
.c.o:
        $(CC) $(CFLAGS) -c $<
```

where the macros are defined as

```
CC=cc
CFLAGS=-V
```

The other built-in macros are:

$*	Target name, minus suffix
$@	Full target name
$<	List of referred files
$?	Referred files newer than target

Each command line *argument* should be a macro definition of the form

```
OBJECT=a.o b.o
```

Arguments that include spaces must be surrounded by quotation marks, because blanks are significant to the shell **sh**.

You can specify macro definitions in the **makefile**, in the environment, or as a command-line argument. A macro defined as a command-line argument always overrides a definition of the same macro name in the environment or in the **makefile**. Normally, a definition in a **makefile** overrides a definition of the same macro name in the environment; however, with the -e option (described below), a definition in the environment overrides a definition in the **makefile**.

Options

The following lists the options that can be passed to **make** on its command line.

-d (Debug) Give verbose printout of all decisions and information going into decisions.

-e Force macro definitions in environment to override those in the **makefile**.

-f *file* *file* contains the **make** specification. If this option does not appear, **make** uses the file **makefile**, which is sought first in the directories named in the **PATH** environmental variable, and then in the current directory. If *file* is '-', **make** uses the standard input; note, however, that the standard input can be used *only* if it is piped.

-i Ignore all errors from commands, and continue processing. Normally, **make** exits if a command returns an error.

-n Test only; suppresses actual execution of commands.

-p Print all macro definitions and target descriptions.

-q Return a zero exit status if the targets are up to date. Do not execute any commands.

-r Do not use the built-in rules that describe dependencies.

-s Do not print command lines when executing them. Commands preceded by '@' are not printed, except under the **-n** option.

-t (Touch option) Force the dates of targets to be the current time, and bypass actual regeneration.

Source File Path

If a file is not specified with an absolute pathname beginning with '/', **make** first looks for the file in the current directory. If the file is not found in the current directory, **make** searches for it in the list of directories specified by macro **$(SRCPATH)**. This allows you to compile a program in an object directory separate from the source directory. For example

```
export SRCPATH=/usr/src/local/me
make
```

or alteratively

```
make SRCPATH=/usr/src/local/me
```

builds objects in the current directory as specified by the **makefile** and sources in **/usr/src/local/me**. To test changes to a program built from several source files, copy only the files you wish to change to the current directory; make will use the local sources and find the other sources on the **$(SRCPATH)**.

Note that **$(SRCPATH)** can be a single directory, as in the above example, or a ':'-separated list of directories, as described in the Lexicon entry for the function **path()**.

Files

makefile
Makefile — List of dependencies and commands
/usr/lib/makeactions — Default actions
/usr/lib/makemacros — Default macros

See Also

as, cc, commands, ld, srcpath, touch
The make Programming Discipline, tutorial

Diagnostics

make reports its exit status if it is interrupted or if an executed command returns error status. It replies "Target *name* not defined" or "Don't know how to make target *name*" if it cannot find appropriate rules.

Notes

The order of items in **makemacros/.SUFFIXES** is significant. The consequent of a default rule (e.g., **.o**) must *precede* the antecedent (e.g., **.c**) in the entry **.SUFFIXES**. Otherwise, **make** will not work properly.

malloc() — General Function (libc)

Allocate dynamic memory
char *malloc(*size***) unsigned** *size***;**

malloc helps to manage a program's free-space arenas. It uses a circular, first-fit algorithm to select an unused block of at least *size* bytes, marks the portion it uses, and returns a pointer to it. The function **free** returns allocated memory to the free memory pool.

Each area allocated by **malloc** is rounded up to the nearest even number and preceded by an **unsigned int** that contains the true length. Thus, if you ask for three bytes you get four, and the **unsigned** that precedes the newly allocated area is set to four.

When an area is freed, its low order bit is turned on; consolidation occurs when **malloc** passes over an area as it searches for space. The end of each arena contains a block with a length of zero, followed by a pointer to the next arena. Arenas point in a circle.

The most common problem with **malloc** occurs when a program modifies more space than it allocates with **malloc**. This can cause later **mallocs** to crash with a message that indicates that the arena has been corrupted.

Example

This example reads from the standard input up to *NITEMS* items, each of which is up to *MAXLEN* long, sorts them, and writes the sorted list onto the standard output. It demonstrates the functions **qsort**, **malloc**, **free**, **exit**, and **strcmp**.

```
#include <stdio.h>
#define NITEMS 512
#define MAXLEN 256
char *data[NITEMS];
char string[MAXLEN];

main()
{
     register char **cpp;
     register int count;
     extern int compare();
     extern char *malloc();
     extern char *gets();

     for (cpp = &data[0]; cpp < &data[NITEMS]; cpp++) {
          if (gets(string) == NULL)
               break;
          if ((*cpp = malloc(strlen(string) + 1)) == NULL)
               exit(1);
          strcpy(*cpp, string);
     }

     count = cpp - &data[0];
     qsort(data, count, sizeof(char *), compare);

     for (cpp = &data[0]; cpp < &data[count]; cpp++) {
          printf("%s\n", *cpp);
          free(*cpp);
     }
     exit(0);
}
```

```
compare(p1, p2)
register char **p1, **p2;
{
     extern int strcmp();
     return(strcmp(*p1, *p2));
}
```

See Also

arena, calloc(), free(), general functions, malloc.h, memok(), realloc(), setbuf()

Diagnostics

malloc returns NULL if insufficient memory is available.

Notes

The commonest error associated with **malloc** is failing to declare it properly. You should always declare **malloc** as returning a pointer to **char**.

malloc.h — Header File

Definitions for memory-allocation functions
#include <sys/malloc.h>

malloc.h defines constants, structures, and macros used with COHERENT's memory-allocation functions. Note that this header does not declare the library's memory-allocation functions.

See Also

header files

man — Technical Information

Manual macro package
nroff -man *file* ...

The **nroff** macro package **man** formats manual pages in the style of the Lexicon. It includes the following macros:

.B	Boldface font.
.BI	Bold/italic alternating fonts.
.BR	Bold/Roman alternating fonts.
.CO	COHERENT.
.DE	Display end.
.DS	Display start.
.DT	Default tabs.
.HE	Help end.
.HP	Hanging paragraph.
.HS	Help start.
.I	Italic font.
.IB	Italic/bold alternating fonts.
.IP	Indented paragraph.
.IR	Italic/Roman alternating fonts.
.LP	Paragraph, flush left.
.PD	Paragraph distance.
.PP	Paragraph, indented.
.RB	Roman/bold alternating fonts.
.RE	Relative indent end.
.RI	Roman/italic alternating fonts.

.RS Relative indent start.
.SH Subheader.
.SM Smaller size.
.TH Define header.
.TP Tagged paragraph.

Files

/usr/lib/tmac.an — Macro package

See Also

ms, nroff, technical information, troff
nroff, The Text Processing Language, tutorial

man — Command

Print Lexicon entries
man [-w] [*topic* ...]

man prints the COHERENT lexicon entries for each specified *topic* on the standard output. It uses **scat** to display text (with the **-s** option to suppress blank lines). With no arguments, **man** prints a list of each available *topic*.

When used with the **-w** option, it prints the path name of the file instead of printing the document itself.

If environmental variable **PAGER** is defined, **man** pipes its output through the command specified in **PAGER**. For example, the **.profile** command line:

```
export PAGER="exec /bin/scat -1"
```

invokes **/bin/scat** with the command-line argument **-1** (the digit one).

Manual-Page Control Files

man uses two control files when processing manual-page requests. File **/usr/man/man.help** contains the **man**'s help message. This includes a list of valid topics and some explanatory text. The second control file, **/usr/man/man.index**, contains index entries for all manual pages on the system. Lines in this text file are of the form:

relative-path-name topic

where *relative-path-name* gives the subdirectory and file in **/usr/man** that hold the manual-page entry, and *topic* gives a manual-page topic associated with this file. For example, entries

```
COHERENT/ascii  ascii
COHERENT/ascii  ASCII
local/chess  chess
```

associate system manual-page **/usr/man/COHERENT/ascii** with either upper- or lower-case spellings of topic **ascii**. Likewise, rules for a user-written chess game are found in file **/usr/man/local/chess** and are retrieved using topic **chess**.

Adding Manual-Page Entries

When writing new manual-page entries for COHERENT, we recommend that you place them in subdirectories of **/usr/man**. These subdirectories should be uniquely named to avoid possible name-space collisions. A good rule of thumb is to name the subdirectory after the application with which it is associated. This also allows them to be updated easily, as all manual-pages associated with a given application reside in a specific subdirectory.

When you add manual-page entries to the system, you should also append a list of topics to **/usr/man/man.help**. In addition, you must append a line to the end of file **/usr/man/man.index** for each newly added topic.

Files

/usr/man/* — Directories that hold manual pages

See Also

commands, help, install, more, PAGER, scat

manifest constant — Definition

A **manifest constant** is a numeric constant that is given a name so it can be defined differently under different computing environments. An example is **EOF**, the end-of-file marker, which has wildly different representations under different operating systems. Note, too, that numerals are manifest constants by definition.

The use of manifest constants in programs helps to ensure that code is portable by isolating the definition of these elements in a single header file, where they need to be changed only once.

See Also

#define, definitions, NULL, portability

math.h — Header File

Declare mathematics functions
#include <math.h>

math.h is the header file to be included with programs that use any of COHERENT's mathematics routines. It includes the following: definitions for mathematical functions; error return values, as used by the **errno** function; definitions of mathematical constants, e.g., **HUGE_VAL**; the definition of structure **cpx**, which describes complex variables; definitions of internal compiler functions; and, finally, declarations of all mathematical functions.

See Also

header files, mathematics library

mathematics library — Overview

The following mathematics routines are available with COHERENT:

acos() . . . . . Calculate inverse cosine
asin() . . . . . Calculate inverse sine
atan() . . . . . Calculate inverse tangent
atan2() . . . . . Calculate inverse tangent of quotient
cabs() . . . . . Calculate complex absolute value
ceil() . . . . . Set numeric ceiling
cos(). . . . . Calculate cosine
cosh() . . . . . Calculate hyperbolic cosine
exp() . . . . . Calculate exponent
fabs() . . . . . Calculate absolute value function
floor() . . . . . Calculate floor function
hypot() . . . . . Calculate hypotenuse
j0() . . . . . Calculate Bessel function, order 0
j1() . . . . . Calculate Bessel function, order 1
jn() . . . . . Calculate Bessel function, order *n*

log() . . . . . Calculate natural logarithm
log10() . . . . . Calculate common logarithm
pow() . . . . . Calculate power
sin() . . . . . Calculate sine
sinh() . . . . . Calculate hyperbolic sine
sqrt() . . . . . Calculate square root
tan(). . . . . Calculate tangent
tanh() . . . . . Calculate hyperbolic tangent

See Also

Lexicon, Libraries, math.h

Press, W.H., Flannery, B.P., Teukolsky, S.A., Vetterling, W.T.: *Numerical Recipes in C*. New York: Cambridge University Press, 1988. *Highly recommended.*

Notes

When programs that contain mathematics routines are compiled, the mathematics libraries must be called specifically on the **cc** command line. For example, to compile the example presented under the entry for **acos**, use the following **cc** command line:

```
cc -f acos.c -lm
```

The **-f** option links in the floating point routines for **printf**, while the **-lm** option links in the mathematics libraries. Note that the **-lm** option must come *last* on the **cc** command line, or the library will not be searched properly.

mboot — Device Driver

Master boot block for hard disk

To be bootable, a COHERENT file system must contain a boot block (either **boot** or **mboot**). In addition, all hard disks must contain the master boot block **mboot** or an equivalent.

mboot is the master boot block for a hard-disk drive. It is compatible with, and therefore can replace, the IBM master boot block installed by the MS-DOS command **FDISK**. It must be installed in the first sector of the hard disk, as follows:

```
/etc/fdisk -b /conf/mboot /dev/at0x
/bin/sync
```

mboot searches its internal partition table (updated by the command **fdisk**) for an active partition. You can select an alternate partition by pressing 0 through 7 before the system selects the active partition. If the selected partition is of non-zero size with a valid partition boot block, COHERENT executes that partition's boot block. Otherwise, the prompt

```
Select partition [0-7]
```

appear, and the system waits for you to select the partition you want.

Files

/conf/mboot — Hard-disk master boot block

See Also

boot, device drivers, fdisk, mkfs

mcmp() — Multiple-Precision Mathematics

Compare multiple-precision integers
#include <mprec.h>
int mcmp(*a*, *b*)
mint **a*, **b*;

The COHERENT system includes a suite of routines that allow you to perform multiple-precision mathematics. The function **mcmp** compares the multiple-precision integers (or **mints**) pointed to by *a* and *b*. It returns a signed integer less than, equal to, or greater than zero according to whether the value pointed to by *a* is less than, equal to, or greater than that pointed to by *b*.

See Also

multiple-precision mathematics

mcopy() — Multiple-Precision Mathematics

Copy a multiple-precision integer
#include <mprec.h>
void mcopy(*a*, *b*)
mint **a*, **b*;

The COHERENT system includes a suite of routines that allow you to perform multiple-precision mathematics. The function **mcopy** sets the multiple-precision integer (or **mint**) pointed to by *b* to the value pointed to by *a*.

See Also

multiple-precision mathematics

mdata.h — Header File

Define machine-specific magic numbers
#include <sys/mdata.h>

mdata.h defines the "magic numbers" for the machine upon which COHERENT is being run.

See Also

header files

mdiv() — Multiple-Precision Mathematics

Divide multiple-precision integers
#include <mprec.h>
void mdiv(*a*, *b*, *q*, *r*)
mint **a*, **b*, **q*, **r*;

The COHERENT system includes a suite of routines that allow you to perform multiple-precision mathematics. The function **mdiv** divides the multiple-precision integer (or **mint**) pointed to by *a* with that pointed to by *b*. It writes the quotient and remainder into, respectively, *q* and *r*. *b* must not be zero. The results of the operation are defined by the following conditions:

1. $a = q * b + r$
2. The sign of *r* equals the sign of *q*

3. The absolute value of *r* is greater than the absolute value of *b*.

See Also

multiple-precision mathematics

me — Command

MicroEMACS screen editor
me [**-e** *errorfile*] [**-f** *bindfile*] [*textfile* ...]

me is the command for MicroEMACS, the screen editor for COHERENT. With MicroEMACS, you can insert text, delete text, move text, search for a string and replace it, and perform many other editing tasks. MicroEMACS reads text from files and writes edited text to files; it can edit several files simultaneously, while displaying the contents of each file in its own screen window.

Screen Layout

Before you can use MicroEMACS, you must set the environmental variable **TERM** in your environment. If you do not set this variable explicitly in your **.profile** file, COHERENT sets it by default to **ansipc**. See the Lexicon entry **TERM** for details.

If the command **me** is used without arguments, MicroEMACS opens an empty buffer. If used with one or more file name arguments, MicroEMACS will open each of the files named, and display its contents in a window. If a file cannot be found, MicroEMACS will assume that you are creating it for the first time, and create an appropriately named buffer and file descriptor for it.

The last line of the screen is used to print messages and inquiries. The rest of the screen is portioned into one or more *windows* in which text is displayed. The last line of each window shows whether the text has been changed, the name of the buffer, and the name of the file associated with the window.

MicroEMACS notes its *current position*. It is important to remember that the current position is always to the *left* of the cursor, and lies *between* two letters, rather than at one letter or another. For example, if the cursor is positioned at the letter 'k' of the phrase "Mark Williams", then the current position lies *between* the letters 'r' and 'k'.

Commands and Text

The printable ASCII characters, from ' ' to '~', can be inserted at the current position. Control characters and escape sequences are recognized as *commands*, described below. A control character can be inserted into the text by prefixing it with **<ctrl-Q>** (that is, hold down the **<control>** key and type the letter 'Q').

There are two types of commands to remove text. *Delete* commands remove text and throw it away, whereas *kill* commands remove text but save it in the *kill buffer*. Successive kill commands append text to the previous kill buffer. Moving the cursor before you kill a line will empty the kill buffer, and write the line just killed into it.

Search commands prompt for a search string terminated by **<return>** and then search for it. Case sensitivity for searching can be toggled with the command **<esc>**@. Typing **<return>** instead of a search string tells MicroEMACS to use the previous search string.

Some commands manipulate words rather than characters. MicroEMACS defines a word as consisting of all alphabetic characters, plus '_' and '$'. Usually, a character command is a control character and the corresponding word command is an escape sequence. For example, **<ctrl-F>** moves forward one character and **<esc>F** moves forward one word.

MicroEMACS can handle blocks of text as well as individual characters, words, and lines. MicroEMACS defines a block of text as all the text that lies between the *mark* and the

current position of the cursor. For example, typing **<ctrl-W>** kills all text from the mark to the current position of the cursor; this is useful when moving text from one file to another. When you invoke MicroEMACS, the mark is set at the beginning of the file; you can reset the mark to the cursor's current position by typing **<ctrl-@>**.

Using MicroEMACS with the Compiler

MicroEMACS can be invoked automatically by the compiler command **cc** to help you repair all errors that occur during compilation. The **-A** option to **cc** causes MicroEMACS to be invoked automatically when an error occurs. The compiler error messages are displayed in one window, the source code in the other, and the cursor is at the line on which the first error occurred. You can correct the errors one by one. To move to the next error in the list, type **<ctrl-X>>**; to move the previous error, type **<ctrl-X><**.

When have finished making corrections, exit from MicroEMACS by typing **<ctrl-Z>**, as usual; the compiler will automatically be re-invoked to re-compile the corrected source code. If more errors are found, MicroEMACS will be re-invoked with the new list of errors. This cycle will continue either until the file compiles without error, or until you break the cycle by typing **<ctrl-U> <ctrl-X> <ctrl-C>**.

The option **-e** to the **me** command allows you to invoke the error buffer by hand. For example, the commands

```
cc myprogram.c 2>errorfile
me -e errorfile myprogram.c
```

divert the compiler's error messages into **errorfile**, and then invokes MicroEMACS to let you correct them interactively.

The MicroEMACS Help Facility

MicroEMACS has a built-in help facility. With it, you can ask for information either for a word that you type in, or for a word over which the cursor is positioned. The MicroEMACS help file contains the bindings for all library functions and macros included with COHERENT.

For example, consider that you are preparing a C program and want more information about the function **fopen**. Type **<ctrl-X>?**. At the bottom of the screen will appear the prompt

```
Topic:
```

Type **fopen**. MicroEMACS will search its help file, find its entry for **fopen**, then open a window and print the following:

```
Open a stream for standard I/O
#include <stdio.h>
FILE *fopen (name, type) char *name, *type;
```

If you wish, you can kill the information in the help window and copy it into your program, to ensure that you prepare the function call correctly.

Consider, however, that you are checking a program written earlier, and you wish to check the call for a call to **fopen**. Simply move the cursor until it is positioned over one of the letters in **fopen**, then type **<esc>?**. MicroEMACS will open its help window, and show the same information it did above.

To erase the help window, type **<ctrl-X>1**.

Options

The following list gives the MicroEMACS commands. They are grouped by function, e.g., *Moving the cursor*. Some commands can take an *argument*, which specifies how often the command is to be executed. The default argument is 1. The command **<ctrl-U>** introduces an argument. By default, it sets the argument to four. Typing **<ctrl-U>** followed by a number sets the argument to that number. Typing **<ctrl-U>** followed by one or more **<ctrl-U>**s multiplies the argument by four.

Moving the Cursor

<ctrl-A> Move to start of line.

<ctrl-B> (Back) Move backward by characters.

<esc>B Move backward by words.

<ctrl-E> (End) Move to end of line.

<ctrl-F> (Forward) Move forward by characters.

<esc>F (Forward) Move forward by words.

<esc>G Go to an absolute line number in a file. Same as **<ctrl-X>G**.

<ctrl-N> (Next) Move to next line.

<ctrl-P> (Previous) Move to previous line.

<ctrl-V> Move forward by pages.

<esc>V Move backward by pages.

<ctrl-X>= Print the current position.

<ctrl-X>G Go to an absolute line number in a file. Can be used with an argument; otherwise, it will prompt for a line number. Same as **<esc>G**.

<ctrl-X>[Go to matching C delimiter. For example, if the cursor is positioned under the character '{', then typing **<ctrl-X>[** moves the cursor to the next '}'. Likewise, if the cursor is positioned under the character }, then typing **<ctrl-X>[** moves the cursor to the first preceding '{'. MicroEMACS recognizes the delimiters [,], {, }, (,), /*, and */.

<ctrl-X>] Toggle reverse-video display of matching C delimiters. For example, if reverse-video displaying is toggled on, then whenever the cursor is positioned under a '}' MicroEMACS displays the first preceding '{' in reverse video (should it be on the screen). MicroEMACS recognizes the delimiters [,], {, }, (,), /*, and */.

<esc>! Move the current line to the line within the window given by *argument*; the position is in lines from the top if positive, in lines from the bottom if negative, and the center of the window if zero.

<esc>< Move to the beginning of the current buffer.

<esc>> Move to the end of the current buffer.

Killing and Deleting

<ctrl-D> (Delete) Delete next character.

<esc>D Kill the next word.

<ctrl-H> If no argument, delete previous character. Otherwise, kill *argument* previous characters.

<ctrl-K> (Kill) With no argument, kill from current position to end of line; if at the end, kill the newline. With argument set to one, kill from beginning of line to current position. Otherwise, kill *argument* lines forward (if positive) or backward (if negative).

<ctrl-W> Kill text from current position to mark.

<ctrl-X><ctrl-O>
Kill blank lines at current position.

<ctrl-Y> (Yank back) Copy the kill buffer into text at the current position; set current position to the end of the new text.

<esc><ctrl-H>
Kill the previous word.

<esc><DEL>
Kill the previous word.

<DEL> If no argument, delete the previous character. Otherwise, kill *argument* previous characters.

Windows

<ctrl-X>1 Display only the current window.

<ctrl-X>2 Split the current window into two windows. This command is usually followed by **<ctrl-X>B** or **<ctrl-X><ctrl-V>**.

<ctrl-X>N (Next) Move to next window.

<ctrl-X>P (Previous) Move to previous window.

<ctrl-X>Z Enlarge the current window by *argument* lines.

<ctrl-X><ctrl-N>
Move text in current window down by *argument* lines.

<ctrl-X><ctrl-P>
Move text in current window up by *argument* lines.

<ctrl-X><ctrl-Z>
Shrink current window by *argument* lines.

Buffers

<ctrl-X>B (Buffer) Prompt for a buffer name, and display the buffer in the current window.

<ctrl-X>K (Kill) Prompt for a buffer name and delete it.

<ctrl-X><ctrl-B>
Display a window showing the change flag, size, buffer name, and file name of each buffer. move cursor out of other window, ^X-N the esc 1

<ctrl-X><ctrl-F>
(File name) Prompt for a file name for current buffer.

<ctrl-X><ctrl-R>
(Read) Prompt for a file name, delete current buffer, and read the file.

<ctrl-X><ctrl-V>
(Visit) Prompt for a file name and display the file in the current window.

Saving Text and Exiting

<ctrl-X><ctrl-C>
Exit without saving text.

<ctrl-X><ctrl-S>
(Save) Save current buffer to the associated file.

<ctrl-X><ctrl-W>
(Write) Prompt for a file name and write the current buffer to it.

<ctrl-Z> Save current buffer to associated file and exit.

Compilation Error Handling

<ctrl-X>> Move to next error.

<ctrl-X>< Move to previous error.

Search and Replace

<ctrl-R> (Reverse) Incremental search backward; a pattern is sought as each character is typed.

<esc>R (Reverse) Search toward the beginning of the file. Waits for entire pattern before search begins.

<ctrl-S> (Search) Incremental search forward; a pattern is sought as each character is typed.

<esc>S (Search) Search toward the end of the file. Waits for entire pattern before search begins.

<esc>% Search and replace. Prompt for two strings; then search for the first string and replace it with the second.

<esc>/ Search for next occurrence of a string entered with the **<esc>S** or **<esc>R** commands; this remembers whether the previous search had been forward or backward.

<esc>@ Toggle case sensitivity for searches. By default, searches are case insensitive.

Keyboard Macros

<ctrl-X>(Begin a macro definition. MicroEMACS collects everything typed until the next **<ctrl-X>)** for subsequent repeated execution. **<ctrl-G>** breaks the definition.

<ctrl-X>) End a macro definition.

<ctrl-X>E (Execute) Execute the keyboard macro.

<ctrl-X>M Bind a newly created keyboard macro to a given keystroke or set of keystrokes.

Flexible Key Bindings

<ctrl-X>R Replace one binding with another.

<ctrl-X>X Rebind the prefix (meta) keys, and the multiple-execution key **<ctrl-U>**.

<ctrl-X>S Prompt for a file name, and write all flexible keybindings and macros into it.

<ctrl-X>L Prompt for a file name, and read all flexible keybindings and macros from it.

<ctrl-X>I Rebind current macro to the initialization macro.

By default, MicroEMACS checks for the existence of file **$HOME/.emacs.rc** and executes it if found. The **-f** option lets you specify an alternate file of keybindings macros from the **me** command line. After loading the file, MicroEMACS then executes the initialization macro, if one exists. For example, to load the keybindings file **bindings** and edit file **textfile**, use the command:

```
me -f bindings textfile
```

Change Case of Text

<esc>C (Capitalize) Capitalize the next word.

<ctrl-X><ctrl-L>
(Lower) Convert all text from current position to mark into lower case.

<esc>L (Lower) Convert the next word to lower case.

<ctrl-X><ctrl-U>
(Upper) Convert all text from current position to mark into upper case.

<esc>U (Upper) Convert the next word to upper case.

White Space

<ctrl-I> Insert a tab. Default behavior is to move the cursor to the nearest 8's boundary; for example, if the cursor is in the 62nd column on the screen, pressing **<ctrl-I>** moves it to column 64.

When used with a positive argument, change the behavior of the tab key. For example, **<ctrl-U>4<ctrl-I>** commands MicroEMACS to insert enough spaces for a tab key to reach a four's boundary.

When used with a negative argument, change the behavior of the tab character. For example, **<ctrl-U>-4<ctrl-I>** says that a tab character on a file will take you to the nearest 4's boundary. Thus, if you have a file with tabs in it and you use '-4', the appearance of the file on the screen will change; but if you use '4' the appearance of the file on the screen will not change.

Exporting the shell variable **TABSIZ=4** will also change the behavior of MicroEMACS this way.

<ctrl-J> Insert a new line and indent to current level. This is often used in C programs to preserve the current level of indentation.

<ctrl-M> (Return) If the following line is not empty, insert a new line; if empty, move to next line.

<ctrl-O> Open a blank line; that is, insert newline after the current position.

<tab> With argument, set tab fields at every *argument* characters. An argument of zero restores the default of eight characters. Setting the tab to any character other than eight causes space characters to be set in your file instead of tab characters.

Send Commands to Operating System

<ctrl-C> Suspend MicroEMACS and execute a subshell. Typing **<ctrl-D>** returns you to MicroEMACS and allows you to resume editing.

<ctrl-X>! Prompt for a shell command and execute it.

These commands recognize the shell variable **SHELL** to determine the shell to which it should pass the command.

Setting the Mark

<ctrl-@> Set mark at current position.

<esc>. Set mark at current position.

<ctrl><space>
Set mark at current position.

Help Window

<ctrl-X>? Prompt for word for which information is needed.

<esc>? Search for word over which cursor is positioned.

<esc>2 Erase help window.

Miscellaneous

<ctrl-G> Abort a command.

<ctrl-L> Redraw the screen.

<ctrl-Q> (Quote) Insert the next character into text; used to insert control characters.

<esc>Q Quote a character by numeric value. When you type this command, MicroEMACS prompts you for a numeric value, in decimal. It then inserts into your text the character whose value you type. This command is useful when you wish to enter characters with the high bit set.

<ctrl-T> Transpose the characters before and after the current position.

<ctrl-U> Specify a numeric argument, as described above.

<ctrl-U><ctrl-X><ctrl-C>
Abort editing and re-compilation. Use this command to abort editing and return to COHERENT when you are using the **-A** option to the **cc** command.

<ctrl-X>H Use word-wrap on a region.

<ctrl-X>F Set word wrap to *argument* column. If argument is one, set word wrap to cursor's current position.

<ctrl-X><ctrl-X>
Mark the current position, then jump to the previous setting of the mark. This is useful when moving text from one place in a file to another.

Diagnostics

MicroEMACS prints error messages on the bottom line of the screen. It prints informational messages (enclosed in square brackets '[' and ']' to distinguish them from error messages) in the same place.

MicroEMACS manipulates text in memory rather than in a file. The file on disk is not changed until you save the edited text. MicroEMACS prints a warning and prompts you whenever a command would cause it to lose changed text.

See Also

commands, ed, elvis, ex, sed, TERM, vi

Notes

Because MicroEMACS keeps text in memory, it does not work for extremely large files. It prints an error message if a file is too large to edit. If this happens when you first invoke a file, you should exit from the editor immediately. Otherwise, your file on disk will be truncated. If this happens in the middle of an editing session, however, delete text until the message disappears, then save your file and exit. Due to the way MicroEMACS works, saving a file after this error message has appeared will take more time than usual.

The current version of MicroEMACS, including source code, is proprietary to Mark Williams Company. The code may be altered or otherwise changed for your personal use, but it may *not* be used for commercial purposes, and it may not be distributed without prior written consent by Mark Williams Company. The source code for MicroEMACS is included with COHERENT, and is kept in directory **/usr/src/local**. You are invited to experiment with source code, to modify existing features or add new ones for your own use.

MicroEMACS is based upon the public domain editor by David G. Conroy.

mem — Device Driver

Physical memory file

The special file **/dev/mem** allows the physical memory of the host computer to be read and written just like an ordinary file. The location where I/O will occur can be positioned to any valid byte address by a call to **lseek**. Note that **ps** and related commands use **/dev/kmem**, which manipulates the kernel's data space.

Commands may examine or change addresses in physical memory. Addresses to use when changing the system itself normally are obtained from the system load module (**/coherent**) name list, so that they always reflect the currently running version of the system.

Files

/dev/mem

See Also

core, device drivers, lseek, ps

Diagnostics

On an error, such as nonexistent memory location, **mem** returns -1.

memccpy() — String Function (libc)

Copy a region of memory up to a set character

#include <string.h>
char *memccpy(*dest, src, c, n***)**
char ****dest*, ****src***; unsigned int** *c, n*;

memccpy copies characters from *src* to *dest*, stopping when either it finds the first occurrence of character *c* or it has copied *n* characters. Unlike the routines **strcpy** and **strncpy**, **memcpy** copies from one region to another. Therefore, it will not halt automatically when it encounters a null character.

memccpy returns a pointer to the first location after character *c* in *dest*, or NULL if character *c* was not found.

See Also

memcpy(), strcpy(), strncpy(), string functions, string.h

Notes

memccpy is not part of the ANSI C Standard. Use of this library routine may restrict portability.

If *dest* and *src* overlap, the behavior of **memccpy** is undefined. *dest* should point to enough reserved memory to hold *n* bytes of data; otherwise, data corruption may result.

memchr() — String Function (libc)

Search a region of memory for a character

#include <string.h>
char *memchr(*region, character, n***)**
char *****region*; **int *character*; **unsigned int** *n*;

memchr searches the first *n* characters in *region* for *character*. It returns the address of *character* if it is found, or NULL if it is not.

Unlike the string-search function **strchr**, **memchr** searches a region of memory. Therefore, it does not stop when it encounters a null character.

Example

The following example deals a random hand of cards from a standard deck of 52. The command line takes one argument, which indicates the size of the hand you want dealt. It uses an algorithm published by Bob Floyd in the September 1987 *Communications of the ACM*.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#define DECK 52

main(int argc, char *argv[])
{
     char deck[DECK], *fp;
     int  deckp, n, j, t;

     if(argc != 2 ||
        52 < (n = atoi(argv[1])) ||
        1 > n) {
             printf("usage: memchr n # where 0 < n < 53\n");
          exit(EXIT_FAILURE);
     }
```

```
        /* exercise rand() to make it more random */
        srand((unsigned int)time(NULL));
        for(j = 0; j < 100; j++)
             rand();

        deckp = 0;
        /* Bob Floyd's algorithm */
        for(j = DECK - n; j < DECK; j++) {
             t = rand() % (j + 1);
             if((fp = memchr(deck, t, deckp)) != NULL)
                  *fp = (char)j;
             deck[deckp++] = (char)t;
        }

        for(t = j = 0; j < deckp; j++) {
             div_t card;

             card = div(deck[j], 13);
             t += printf("%c%c  ",
                  /* note useful string addressing */
                  "A23456789TJQK"[card.rem],
                  "HCDS"[card.quot]);

             if(t > 50) {
                  t = 0;
                  putchar('\n');
             }
        }

        putchar('\n');
        return(EXIT_SUCCESS);
}
```

See Also

strchr(), string functions, string.h

memcmp() — String Function (libc)

Compare two regions

#include <string.h>
int memcmp(*region1, region2, count***)**
char ****region1*; char ****region2***; unsigned int** *count***;**

memcmp compares *region1* with *region2* character by character for *count* characters.

If every character in *region1* is identical to its corresponding character in *region2*, then **memcmp** returns zero. If it finds that a character in *region1* has a numeric value greater than that of the corresponding character in *region2*, then it returns a number greater than zero. If it finds that a character in *region1* has a numeric value less than less that of the corresponding character in *region2*, then it returns a number less than zero.

For example, consider the following code:

```
char region1[13], region2[13];
strcpy(region1, "Hello, world");
strcpy(region2, "Hello, World");
memcmp(region1, region2, 12);
```

memcmp scans through the two regions of memory, comparing **region1[0]** with **region2[0]**, and so on, until it finds two corresponding "slots" in the arrays whose contents differ. In the above example, this will occur when it compares **region1[7]** (which contains 'w') with **region2[7]** (which contains 'W'). It then compares the two letters to see which stands first in the character table used in this implementation, and returns the appropriate value.

memcmp differs from the string comparison routine **strcmp** in a number of ways. First, **memcmp** compares regions of memory rather than strings; therefore, it does not stop when it encounters a null character.

Also, **memcmp** can be used to compare an **int** array with a **char** array is permissible because **memcmp** simply compares areas of data.

See Also

strcmp(), string functions, string.h

memcpy() — String Function (libc)

Copy one region of memory into another

#include <string.h>
char *memcpy(*region1*, *region2*, *n***)**
char ***region1*; char ***region2***; unsigned int** *n***;**

memcpy copies *n* characters from *region2* into *region1*. Unlike the routines **strcpy** and **strncpy**, **memcpy** copies from one region to another. Therefore, it will not halt automatically when it encounters a null character.

memcpy returns *region1*.

See Also

strcpy(), string functions, string.h

Notes

If *region1* and *region2* overlap, the behavior of **memcpy** is undefined. *region1* should point to enough reserved memory to hold *n* bytes of data; otherwise, code or data will be overwritten.

memmove() — String Function (libc)

Copy region of memory into area it overlaps

#include <string.h>
char *memmove(*region1*, *region2*, *count***)**
char ***region1*, char ***region2***, unsigned int** *count***;**

memmove copies *count* characters from *region2* into *region1*. Unlike **memcpy**, **memmove** correctly copies the region pointed to by *region2* into that pointed by *region1* even if they overlap. To "correctly copy" means that the overlap does not propagate, not that the moved data stay intact. Unlike the string-copying routines **strcpy** and **strncpy**, **memmove** continues to copy even if it encounters a null character.

memmove returns *region1*.

See Also

string functions, string.h

Notes

region1 should point to enough reserved memory to hold the contents of *region2*. Otherwise, code or data will be overwritten.

memok() — General Function (libc)

Test if the arena is corrupted
memok();

The library function **memok** checks to see if the area has been corrupted. It returns one if the arena is sound, and zero if it has been corrupted.

Example

The following example purposely corrupts the arena, to demonstrate **memok**. Please note that this is not a recommended programming practice.

```
extern char *malloc();
main()
{
     char *p;

      p = malloc(2);                    /* get 2 bytes of memory */
      printf("Arena is %s\n", memok() ? "OK" : "bad");
      strcpy(p, "too long");            /* clobber memory */
      printf("Arena is %s\n", memok() ? "OK" : "bad");
}
```

See Also

arena, calloc(), general functions, malloc(), realloc()

memory allocation — Technical Information

The following diagram shows how COHERENT allocates memory.

Data Segment (maximum size 64 kilobytes)

```
=======================
|      ARENA AND      |
|     FREE MEMORY     |
|=====================| ← SP starts here
|        STACK        |
|=====================|
|   UNITIALIZED DATA  |
|        (bssd)       |
|=====================|
| PRIVATE DATA (prvd) |
|=====================|
|  SHARED DATA (shrd) |
======================= ← DS ES SS point here
```

Code Segment (maximum size 64 kilobytes)

```
=======================
|        CODE         |
|=====================|
|   RUNTIME STARTUP   |
====================== ← CS points here
```

Note that COHERENT can relocate the code and data segments at its own convenience and merely repoint the required segment registers.

The stack *descends* from the highest address in its space toward the static data area; new arguments are placed on the stack in its *lowest* address. Everything from the top of the stack space to the end of the data segment is free to accept dynamically allocated data.

The size of the stack cannot be altered while a program is running. By default, the runtime startup sets the stack size to four kilobytes (4,096 bytes). Note, however, that a highly recursive function may cause the stack to grow larger than four kilobytes so that it overwrites other data areas. This will cause your program to work incorrectly. To reset the amount of stack allocated to a program, use the command **fixstack**.

See Also

data formats, fixstack, technical information

memset() — String Function (libc)

Fill an area with a character
#include <string.h>
char *memset(*buffer*, *character*, *n***)**
char ***buffer*; int** *character***; unsigned int** *n***;**

memset fills the first *n* bytes of the area pointed to by *buffer* with copies of *character*. It casts *character* to an **unsigned char** before filling *buffer* with copies of it.

memset returns the pointer *buffer*.

See Also

string functions, string.h

mesg — Command

Permit/deny messages from other users
mesg [ny]

Normally, a user can communicate with other users by using the commands **msg** and **write**.

In certain situations, it is useful to suppress messages from other users. Therefore, COHERENT supplies the command **mesg**, which, lets you permit or suppress messages from other users. The argument **y** allows messages, whereas argument **n** disallows messages. With no argument, **mesg** tells you whether you can receive messages (as **yes** or **no**) without changing the message state.

Files

/dev/*

See Also

commands, msg, write

Notes

The owner-execute mode bit of the user's **tty** indicates whether messages are allowed.

min() — Multiple-Precision Mathematics

Read multiple-precision integer from stdin
#include <mprec.h>
void min(*a*)
mint **a*;

The COHERENT system includes a suite of routines that allow you to perform multiple-precision mathematics. The function **min** reads a multiple-precision integer (or **mint**) from the standard input and writes it at the address held by *a*. The base of the **mint** is indicated by the value held in the external variable **ibase**.

min accepts leading blanks and an optional leading minus sign; the number is terminated by the first non-legal digit.

See Also

multiple-precision mathematics

minit() — Multiple-Precision Mathematics

Condition global or auto multiple-precision integer
#include <mprec.h>
void minit(*a*)
mint **a*;

The COHERENT system includes a suite of routines that allow you to perform multiple-precision mathematics. The function **mint** helps to create a multiple-precision integer (or **mint**). If a new **mint** is declared to be global or automatic, you must call **minit** before using the variable. This prevents garbage values in the newly created **mint** structure from causing chaos. A **mint** conditioned by **minit** has no value; however, it may be used to receive the result of an operation.

See Also

multiple-precision mathematics

minor number — Definition

Device numbering

A *minor number* specifies the device or type of device to use. COHERENT uses the minor number of a given device in a driver-specific manner. For example, a hard-disk driver may use the minor number to select a disk drive and partition.

Every COHERENT device has a device number associated with it. It is of type **dev_t**, as defined in **<sys/types.h>**. The macro **minor()** in **<sys/stat.h>** extracts the minor number from a given device number.

See Also

device drivers, major number, stat.h

mintfr() — Multiple-Precision Mathematics

Free a multiple-precision integer
#include <mprec.h>
void mintfr(*a*)
mint **a*;

The COHERENT system includes a suite of routines that allow you to perform multiple-precision mathematics. The function **itom** creates a multiple-precision integer (or **mint**). You can call the function **mintfr** to free the storage used by a **mint**.

See Also

multiple-precision mathematics

misc — Technical Information

Library of miscellaneous functions

misc is a collection of library routines. These routines are useful for handling such programming tasks as allocation of memory, copying of strings, displaying variables from C with COBOL-like "picture" descriptions, and supporting virtual arrays via secondary storage.

Source code for the library is kept in the compressed **tar** archive file **/usr/src/misc.tar.Z**. To extract the files into a new subdirectory called **misc**, use the command:

```
zcat /usr/src/misc.tar.Z | tar xvf -
```

To build the library, type the following:

```
cd misc
make
```

For a full description of each function, consult the included **Read_me** file.

Files

/usr/src/misc.tar.Z — Compressed **tar** archive of sources

See Also

tar, technical information, zcat

Notes

The **misc** library is provided on an *as-is* basis only. *Caveat utilitor!*

mitom() — Multiple-Precision Mathematics

Reinitialize a multiple-precision integer
#include <mprec.h>
void mitom(*n*, *a*)
mint *****a; **int** n;

The COHERENT system includes a suite of routines that allow you to perform multiple-precision mathematics. The function **mitom** reinitializes the existing multiple-precision integer (or **mint**) pointed to by a to n.

See Also

multiple-precision mathematics

mkdir — Command

Create a directory
mkdir [**-r**] *directory*

mkdir creates *directory*. Files or directories with the same name as *directory* must not already exist. **directory** will be empty except for the entries '.', the directory's link to itself, and '..', its link to its parent directory. The option **-r** creates directories recursively. For example, the command

```
mkdir -r /foo/bar/baz
```

creates directory **foo** in /; then creates directory **bar** in the newly created directory **foo**; and finally creates directory **baz** in the newly created directory **bar**.

See Also

commands, rm, rmdir

Diagnostics

mkdir fails and prints an error message if you do not have permission to write into directory in which you are attempting to create a new directory, or if the directory in which you attempted to create a new directory does not exist.

mkfnames — Command

Generate data base of user names
mkfnames [*namefile* ...]

mkfnames reads the contents of *namefile* and writes to the standard output a sorted data base that the command **mail** can use as its data base of full users' names. The command **nptx** defines the format of an input line.

If no *namefiles* are named on the command line, **mkfnames** reads the file **/etc/passwd**, and attempts to parse its contents into the form required by **nptx**.

mkfnames is usually used to generate **/usr/lib/mail/fullnames**. If more than one login account has the same part of a name (i.e., the same last name), the lexicographically first login name will be used.

See Also

commands, mail, nptx

mkfs — Command

Make a new file system
/etc/mkfs [**-b** *boot*] [**-d**] [**-f** *name*] [**-i** *inodes*] [**-m** *arg*] [**-n** *arg*] [**-p** *pack*] *filesystem proto*

mkfs makes a new file system. *filesystem* names the file (normally a block special file) where the new file system will reside. The contents of the newly created file system are described in *proto*. *proto* can be either a number or a file name.

If *proto* is a number, **mkfs** creates an empty file system (containing only a root directory) of the size in blocks given by *proto*. The number of i-nodes is calculated as a percentage of this number. The command

```
/etc/mkfs /dev/fha0 2400
```

creates a file system on a high-density, 5.25-inch diskette in drive 0. If the disk is a high-density, 3.5-inch diskette, use the command:

```
/etc/mkfs /dev/fva0 2880
```

If *proto* is a file name, however, the contents of that file will be used as a prototype for modeling the new file system. This prototype file must be laid out in the following manner:

```
bootstrap_file_name file_system_name device_name
no._of_blocks no._of_i-nodes n m
%b XX XX XX
...
directory_name
        directory_name mode user_id group_id contents
        ...
        $
$
```

Each line is described below.

The first line has three fields. Field 1, *bootstrap_file_name*, contains the name of a file that holds the boot strap, which must fit into block 0 of the disk. Field 2, *file_system_name*, gives the name of the file system; and field 3, *device_name*, gives the name of file system's physical device (for example, **/dev/hd1**). Only the first six characters in fields 2 and the first 11 in field 3 are significant; all characters after them are ignored.

The second line contains four fields. Field 1, *no._of_blocks*, gives the size of the file system in blocks; field 2, *no_of_i-nodes*, gives the number of i-nodes in the file system. Because each file or directory requires one i-node, this number represents the limit on the number of files that may be created in the file system. A ratio of seven blocks per i-node generally works well.

Fields 3 and 4 control free list interleaving on your disk. *n* is the size of a "virtual cylinder": **fsck** allocates all the blocks on one virtual cylinder before it advances to the next virtual cylinder. The value of *n* must be less than or equal to 255, and should evenly divide the actual size of a cylinder on the device. *m* tells the system how many blocks to skip each time it increments a free list block number, i.e., the free list "interleave"; *n* mod *m* must be zero. Choosing an optimal interleave value may improve system performance for the device. The optimal values for *n* and *m* are hardware-specific and can be determined by experimentation.

Next, the third line and following begin with **%b**. These list the bad blocks on your storage device. One or more block numbers may appear on each line, separated by white space. These blocks are allocated to the bad block file (i-node 1).

The remaining lines in the *proto* file define the names, modes, and contents of the directories and files in the file system. These lines are divided into fields separated by white space (blanks or tabs) as follows:

- The first field names the file or directory to be created. This field is missing on the first line, which describes the root directory of the file system.
- The second field describes the mode of the file, which is six characters long. The first character gives the file type, that is, whether the file is ordinary ('-'), directory ('d'), block special ('b'), or character special ('c'). The second character is 'u' for set user id on execution, and '-' otherwise. The third character is 'g' for set group id on execution, and '-' otherwise. Characters 4 through 6 specify permissions in octal; for example, **644** specifies read and write permission for the owner, read permission for other users from the same group, and read permission for users from other groups.
  If the above file type were a directory, subsequent files are recursively defined under that directory, until the current level of directory is terminated by a line containing a '$' character.

- The next two fields specify the owner's numerical user id and group id.
- The last field describes file contents. For a directory, it is not needed. For an ordinary file, it is the name of a COHERENT file that will be copied into the newly created file. For block or character-special files, there are two fields that specify the numbers of the major and minor devices.

Finally, each directory's description and the entire *proto* file must terminate with dollar signs '$'.

The *proto* file need not contain all of the above fields. However, it must contain the name of the boot block (line 1), the number of blocks and the number of i-nodes (line 2), the list of bad blocks, the name of at least one directory, and the dollar sign that ends the file.

Command-line Options

mkfs recognizes the following command-line options:

-b *boot*
Specifies the file to use as the "bootstrap" for the filesystem.

-d Preserve file dates and times on the new file system.

-f *name*
Label the filesystem with the given *name*. *name* must be less than seven characters in length.

-i *inodes*
Use *inodes* as the number of inodes for the filesystem.

-m *arg*
Set the number of blocks to skip when incrementing virtual block number. This is the same as the *m* option as set on line 2 of the prototype file. You can use this option if you choose not to use a prototype file.

-n *arg*
Set the size of size of a "virtual cylinder". This is the same as the *n* option as set on line 2 of the prototype file. You can use this option if you choose not to use a prototype file.

-p *pack*
Set the filesystem "pack name" to *pack*. *pack* must be less than seven characters in length.

Example

The following example specifies a proto file for a high-density, 5.25-inch floppy disk; note that this floppy disk is faulty and contains a number of bad blocks:

```
/conf/boot.fha
2400 100
%b 55
%b 185 86
d--755 3 1
       coherent ---644 3 1 /coherent
       tmp        d--777 3 1
       $
       bin        d--755 3 1
                  mail      -u-755 0 1 /bin/mail
       $
       dev        d--755 3 1
                  tty30     c--644 0 1 3 0
                  tty35     c--644 0 1 3 5
                  mt0       b--600 0 1 12 0
       $
$
```

You can use the command **badscan** to draw up the list of bad blocks on your disk and create a skeleton *proto* file.

See Also

badscan, chmod, commands, fsck, mount, restor, unmkfs

Diagnostics

Diagnostic message are generated for badly constructed *proto* files or for I/O errors on the file system.

mknod() — COHERENT System Call (libc)

Create a special file
#include <sys/ino.h>
#include <sys/stat.h>
mknod(*name, mode, addr*)
char **name*; **int** *mode, addr*;

mknod is the COHERENT system call that creates a special file. A *special file* is one through which a device is accessed, or a named pipe.

mode gives the type of special file to be created. It can be set to **IFBLK**, for a block-special device, such as a disk driver; to **IFCHR**, for a character-special device, such as a serial-port driver; to **IFDIR**, for a directory; or to **IFPIPE**, for a named pipe.

address is a parameter interpreted by the driver; it might specify the channel of a multiplexor or the unit number of a drive. Note that this is not used with named pipes.

See Also

COHERENT system calls, named pipe, pipe

Notes

Only the superuser **root** can use the system call **mknod**; this is a security feature.

mknod — Command

Make a special file or named pipe
/etc/mknod [**-f**] *filename type major minor*
/etc/mknod [**-f**] *filename* **p**

In the first form, **mknod** creates a *special file*, which provides access to a device by the *filename* specified. Special files are conventionally stored in the **/dev** directory.

type can be either 'b' (for block-special file) or 'c' (for character-special file). Block-special files tend to be devices such as disks or magnetic tape, upon which COHERENT uses an elaborate buffering strategy. Character-special files are unstructured (character at a time) devices such as terminals, line printers, or communications devices. Character-special files may also be random-access devices; this circumvents system buffering, allowing transfers of arbitrary size directly between the user and the hardware.

The *major* device number uniquely identifies a device driver to COHERENT. The *minor* device number is a parameter interpreted by the driver; it might specify the channel of a multiplexor or the unit number of a drive.

The caller must be the superuser.

In the second form, **mknod** creates a named pipe with the given *filename*. Named pipes can be used for communication between processes.

The **-f** option to **mknod** forces the creation of a new node, even if one of the same name already exists.

Files

/dev/*

See Also

commands, mount

mktemp() — General Function (libc)

Generate a temporary file name
char *mktemp(*pattern***) char ***pattern***;**

mktemp generates a unique file name. It can be used, for example, to name intermediate data files. *pattern* must consist of a string with six **X**'s at the end. **mktemp** replaces these **X**'s with the five-digit process id of the requesting process and a letter that is changed for each subsequent call. **mktemp** returns *pattern*. For example, the call **mktemp("/tmp/sortXXXXXX");** might return the name **/tmp/sort01234a**. It is normal practice to place temporary files in the directory **/tmp**. The start of the file name identifies the originator of the file.

See Also

general functions, getpid(), tempnam(), tmpnam()

mneg() — Multiple-Precision Mathematics

Negate multiple-precision integer
#include <mprec.h>
void mneg(*a, b***)**
mint ***a, ***b;

The COHERENT system includes a suite of routines that allow you to perform multiple-precision mathematics. The function **mneg** negates the value of the multiple-precision integer (or **mint**) pointed to by *a*, and writes the result into the **mint** pointed to by *b*.

See Also

multiple-precision mathematics

mnttab.h — Header File

Structure for mount table
#include <mnttab.h>

mnttab.h defines the structure for the mount table maintained by the functions **/etc/mount** and **/etc/umount**.

See Also

header files, mount, umount

modem — Technical Information

The word *modem* is an abbreviation for "modulation/demodulation device". With the COHERENT system, you can attach a modem to your computer either to dial out for remote communication, to let others dial into your COHERENT system, or both. With your modem, too, you can use COHERENT's UUCP commands to exchange mail and files with remote sites automatically, and to download news and files from the USENET.

This article gives a summary of how to connect your modem to your computer and describe it to the COHERENT system. It also discusses some problems that may crop up when you attempt to use your modem.

Internal vs. External Modems

You can use internal and external modems with COHERENT. An internal modem is, however, more difficult to diagnose problems because there are typically no status lights to indicate operation.

Hooking up a Modem

A modem must be hooked up to a serial port on your computer. To plug your modem into the computer, simply take a normal serial-port cable, one with an RS-232 plug of the appropriate gender at each end, plug one end into your modem and the other into the serial port you wish to use. The Lexicon article **RS-232** describes the wiring of the RS-232 plug in detail; but if you are not skilled with a soldering iron, you are well advised simply to purchase a cable from your local electronics store and be done with it.

Serial Ports

The COHERENT system supports up to four serial ports; the devices for these are named **/dev/com1r** through **/dev/com4r**. If you are not sure which port you have plugged your modem into, perform the following test: First, turn on the modem. Then, type the following command:

```
echo FOO >/dev/com1l
```

If the "TX" light on the modem blinks, then you know the modem is plugged into **COM1**. If it does not, try the command again for **/dev/com2l**, and so on through **com4l** until you find the appropriate port. If no command works, check the wiring on your cable and make sure that the plugs are securely inserted.

Once you have established which port your modem is plugged into, link the device **/dev/modem** to it, using the following command:

```
ln -f /dev/com?l /dev/modem
```

where ? gives the number of the port, 1 through 4.

Edit /etc/ttys

The next step is to edit the file **/etc/ttys** to tell COHERENT how you want the port handled. You must know (1) whether you want the port enabled or disabled; (2) the baud rate of the port (as set by your modem); and (3) the name of the port (which you just determined).

If a port is enabled, remote users can log into the system, either via a terminal directly plugged into the port or via a modem. COHERENT sends a login prompt to every enabled port. The COHERENT system also restricts permissions on all enabled serial ports, so that only the superuser **root** can read and write to the port. This prevents other users who may be using the system from accessing the serial port. If a port is disabled, you can dial out or use a direct-connect UUCP connection via that disabled port.

To dial out on an enabled port, you must first use the command **disable** to disable the port. When you have finished dialing out, run the command **enable** to re-enable the port. Before you can use these commands with a port, the port must first be described in the file **/etc/ttys**.

See the Lexicon article on **ttys** for details on how to edit this file. Note that a modem is a remote device, and must be so described in **/etc/ttys**, or it will not work correctly.

After you have made your changes, type the command

```
kill quit 1
```

to make COHERENT re-read **/etc/ttys** and implement your changes.

Remote-Access Password

If you intend to let people dial into your computer, you are well advised to set the remote-access password. This will require that people who dial in know a special password in addition to whatever password their personal account may have. To set this password, log in as the superuser **root**; then enter the command

```
passwd remacc
```

This will walk you through setting a password for user **remacc**, which is the remote-access account.

Edit /etc/modemcap

Next, you must check the file **/etc/modemcap** and see if that file holds a description that matches your modem. **modemcap** is used by a number of programs to control access to modems, and this file comes with descriptions for many commonly used modems. You find, however, that you must edit an existing entry to match your modem's features exactly; for example, the existing entry may assume that you have a Touch-Tone telephone whereas you actually have a pulse telephone. The Lexicon entry on **modemcap** will walk you through this process. When you have completed editing this entry, note it down, for you will need to insert it elsewhere.

Edit /usr/bin/modeminit

Next, check the file **/usr/bin/modeminit**. This program is read by programs that manipulate the modem, to re-initialize it after use. You must decide how you want the modem to be re-initialized. Basically, if you wish to have people dial into your system, you turn on the modem's auto-answer feature; and you must turn off echoing and the printing of result codes. The commands to use will vary from modem to modem; see the documentation that comes with your modem for details. See below for details on modifying this script.

Edit L-devices

If you intend to use your modem with UUCP, you must insert an entry for it into your the file **/usr/lib/uucp/L-devices**. See the Lexicon entry **L-devices** for details.

Modem Maladies

The rest of this article discusses problems that have arisen with remote login via modem, as diagnosed by the technical support staff of Mark Williams Company.

Difficulty in logging in from a remote site via modem can be the result of problems in one or more of the following: cabling; enabling/disabling the port; flaws in the contents of file **/etc/ttys**; incorrect configuration of the modem; and setting the port to an incorrect state. See Lexicon articles **terminal** and **UUCP** for additional information. The following paragraphs discuss the above-named items in detail.

RS-232 Cabling

When attaching an external modem to your computer, it is important to use a modem cable that supports "full modem control". COHERENT relies on modem-control signals when operating a modem for remote access purposes. When attaching a terminal directly to a serial port, a "null modem" cable must be used. When attaching a modem, a "straight through" cable must be used. See Lexicon articles **RS-232** and **terminal** for further details on cabling.

Enabled vs. Disabled Ports

A serial port can be either enabled or disabled for remote access. Enabling a port allows a user on a remote terminal or modem to log into your COHERENT system. Disabling a port permits a user to dial out or use a direct connect UUCP connection via that disabled port.

If a port is enabled for remote logins and you will use it to call out, you must use the command **disable** to disable the port before you access the port. The commands **uucp** and **mail** automatically disable and re-enable a port.

The port name supplied to an **enable** or **disable** command must *exactly* match the last part of a line in the **/etc/ttys** file (see below). For example, for the command **enable com2pr** to work, there must be an entry in the file **/etc/ttys** which ends with **com2pr**.

When a port is enabled, (1) the first character for the port in file **/etc/ttys** is set to a '1' (one), (2) the permissions for the port are changed so that only the superuser **root** can read and write to the port, preventing other users on the system from accessing the port while a remote session is in progress, and (c) a login prompt is sent to the port.

/etc/ttys

This file should have permissions of 644 (-rw-r--r--) and belong to owner and group **root**. Review the Lexicon entry for **ttys** to ensure that the format of your version of **/etc/ttys** is correct.

Leaving blanks at the end of a line in **/etc/ttys** usually results in error messages stating that a device could not be found.

You do not need to edit the initial '0' or '1' in entries in **/etc/ttys**; this digit is updated by the commands **enable** and **disable**. See the Lexicon entries for **enable** and **disable** for more information.

Another problem can arise when using a variable-speed modem and specifying the baud rate as '3' in file **/etc/ttys**. The '3' setting represents a 2400-1200-300 baud variable-speed port. The problem appears to be that while a modem defaults to 2400 baud, COHERENT may actually be set at one of the other speeds. The answer to this probem is to run the command **modeminit**.

Constant Flickering

Another problem is a constant flickering of send/receive LEDs and an unexplained continual access of the hard drive. This occurs when the port is enabled and the modem is set in echo mode: COHERENT sends the login prompt to the modem, the modem echoes it back to COHERENT, COHERENT then thinks the modem is trying to talk to it and sends the password prompt, and so on *ad infinitum*. To fix this problem, place the modem into no-echo mode, and turn off the display of result codes. The following section discusses this in more detail.

Modem Configuration

A modem that fails to answer an incoming call, hangs up before locking onto the remote carrier, becomes stuck in a loop echoing characters sent to it from the computer, or fails to operate at the expected baud rate probably is configured improperly. To remedy this situation, send the appropriate control string to the modem.

We offer some guidelines here for modem settings. Be warned, however, that modems from different manufacturers usually behave differently, regardless of claims of Hayes compatibility, and that you will need to check the manual for your particular modem.

- Echo should be OFF (usually by setting "E0").
- Result codes should be OFF (usually by setting "Q1").
- Modem status "DCD" should follow true carrier detect status, rather than being always on (usually by setting "&C1").
- Auto answer should be ON (usually obtained by setting register S0 to a nonzero value equal to the number of rings before answer).
- The delay value for "Wait for Carrier/Dial Tone" (usually register S7) should not be too short.

The scripts below show typical initialization for "Hayes-compatible" and Trailblazer modems. They are only examples; your modem may need something different. Please note that the commands **sleep** and **stty** are necessary in the first example so that the command string will be sent to the modem at 2400 baud; otherwise, the string is sent at the default port speed, which is 9600 baud.

```
# initialize 2400-baud Hayes-compatible modem
disable com3r
sleep 3 > /dev/com3l &
stty 2400 > /dev/com3l
echo 'AT E0 Q1 V0 S0=1 &C1 M3' > /dev/com3l
sleep 3
enable com3r

# initialize 9600 baud internal Trailblazer on com2
/etc/disable com2r
cat > /dev/com2l << EOF
at
at e0 t v0 x3 h0
at s0=1 s7=60 s48=1 s51=252 s52=0 s54=3 s58=2
at
EOF
/etc/enable com2r
```

You can edit the file **/usr/bin/modeminit** to suit your modem. To ensure that your modem is initialized every time you start COHERENT, you should add a line saying

```
/usr/bin/modeminit
```

to your copy of the file **/etc/rc**.

See Also

modemcap, modem control, RS-232, technical information, terminal

Notes

One final bit of hard-won wisdom: once you have something working, write down what you did, and store it in a place where you won't lose it. It makes life easier just knowing that you're looking for a female-to-female cable instead of male-to-female or male-to-male.

modemcap — System Maintenance

Modem-description language
/etc/modemcap

modemcap is a language for describing modems to your system. It resembles the **termcap** language in its syntax, although the two are by no means identical. With **modemcap**, you can describe your modem to any program that automatically dials out on your modem; this should spare you the tedium of continually describing your modem to one program after another.

The copy of **/etc/modemcap** included with your release of COHERENT already contains descriptions of many popular modems; the chances are good that yours has already been described for you.

Each **modemcap** command is one of three types: *flag*, *string*, or *number*. A *flag* command signals that your modem is performs a particular action or has a particular feature. A *string* command gives the command that your modem recognizes to perform a particular action. For example, many modems recognize that the string **at** means that you want to gain its attention. Finally, a *number* command sets a value or parameter for your modem, such as the highest baud rate it recognizes.

The following table describes each **modemcap** command:

Name	*Type*	*Meaning*
ad	number	Delay after **as**
as	flag	Numbers are in ASCII, not binary
at	string	Attention string, forces model into command mode from online mode
bd	number	Highest online baud rate
bl	number	Alternate lower baud rate
ce	string	Command end string (required if CS is present)
cl	string	String from modem on remote connection at BL baud rate
co	string	String from modem on remote connection at BD baud rate
cs	string	Command start string
de	string	End dial command string (required if DS is present)
di	flag	Modem has a dialer
ds	string	Start dial command string
id	number	Delay after IS
is	string	Initialization string, resets modem to offline, ready to dial
hc	flag	Modem hangs up when DTR drops

hu	string	Hangup command
tt	flag	Modem dials touchtone by default (or DS is set that way)

All commands, such as **ds** (dial command) and **hu** (hang up) will be prefixed by **cs** and ended with **ce**. If there is a common prefix and suffix, use this feature. Otherwise, each command will have to have the entire string built in.

Example Entry

The following gives the entry in **/etc/modemcap** for the Hayes Smartmodem 1200:

```
hy|hayes|Hayes Smartmodem 1200:\
      :as:at=+++:ad#3:bd#1200:bl#300:cs=AT:ce=\r:co=CONNECT:\
      :cl=CONNECT:di:ds=DT :de=:is=ATQ0 V1 E1\r:id#2:\
      :hc:hu=H0 V0 E0 Q1:tt:
```

Each field is separated by a colon. A backslash '\' character at the end of each line but the last lets the description extend over more than one line.

The three fields gives three versions of the modem's name, separated by vertical bars '|'. The first version of the name is a two-character mnemonic; this must be unique. The other two versions give fuller versions of the name; these are optional.

The following explains each field in detail:

as	Numbers are in binary mode.
at=+++	To gain the attention of the modem, type **+++**.
ad#3	Delay three milliseconds after a number.
bd#1200	Maximum baud rate is 1200.
bl#300	Minimum baud rate is 300.
cs=AT	To initiate a command string, type **AT**.
ce=\r	A command string is ended by a carriage-return character.
co=CONNECT	Modem returns the string **CONNECT** when it makes a connection at 1200 baud.
cl=CONNECT	Modem returns the string **CONNECT** when it makes a connection at 300 baud.
di	The modem can dial a telephone number.
ds=DT	Begin dialing, touch-tone mode.
de=	No special string is needed to end the dial string.
is=ATQ0	To initialize the modem, type **ATQ0 V1 E1<return>**.
id#2	Delay two seconds after entering the initialization string.
hc	The modem hangs up when DTR drops (i.e., it hangs up when the program requests a hangup).
hu=H0	To hang up, type **H0 V0 E0 Q1**.

tt	The modem dials touch-tone by default.

Currently Recognized Modems

The file **/etc/modemcap** includes descriptions of the following modems:

tbfast	Trailblazer, 9600 baud
xtb2400	Trailblazer, 2400 baud
hayes	Hayes Smartmodem 1200
avatex	Avatex 2400 (clone of Hayes Smartmodem 2400)
promodem	Prometheus Promodem 1200
mk12	Signalman Mark XII
dc300	Radio Shack Direct-Connect 300 Modem

See Also

system maintenance, termcap

modem control — Technical Information

This article documents COHERENT's *modem control* protocol. Modem control describes how COHERENT handles RS-232 signals other than Receive Data and Transmit Data. The behavior of COHERENT's suite of device drivers for serial devices is evolving; changes will be documented in further revisions to this manual.

Many processes can have a device open at the same time. *First open* occurs if a process opens a device when no process has opened the device. *Last close* occurs when a process closes the port and no other remaining process has the port open.

al[01] Drivers (Devices com[1-4]*)

On first open, RTS and DTR are asserted by the computer, regardless of whether the specified device used modem control. If modem control is used (the high-order bit in minor number set to zero), **open()** does not complete until CD is true. Once an **al[01]** device has been opened with modem control, loss of CD to that port causes **SIGHUP** to be sent to all processes in the group keeping the port open.

hs Driver (Devices hs0[0-7]{r})

Unfortunately, in this driver the meaning of the high-order bit in minor device numbers is reversed from that of the **al[01]** drivers: a one in this bit position indicates modem control *is* used.

Setting the speed of an **hs** device to a nonzero value causes assertion of RTS and DTR, regardless of whether the device has modem control; this happens whenever the device is opened, and at other times. Setting the speed to zero deasserts RTS and DTR.

If modem control is used (high-order bit in minor number is set to one), **open()** completes regardless of other modem signals, but attempts to read or write the device fail until DSR from the modem is true, with **errno** set to **EIO**. Transition to **false** of DSR causes **SIGHUP** to be sent to the process group using the device. In addition, when modem control is used, the computer halts transmission to the port whenever CTS from the modem goes false.

See Also

modem, RS-232, technical information

modeminit — System Maintenance

Initialize a modem
/usr/bin/modeminit

The script **/usr/bin/modeminit** can be used to initialize a modem. In its default form, this script sets a Hayes-compatible modem into no-echo mode, turns off command responses, and turns on auto-answering.

You can edit the file **rc** to have it call this script. This will ensure that your modem is properly initialized when you boot COHERENT.

See Also

modem, system maintenance

modf() — General Function (libc)

Separate integral part and fraction
double modf(*real*, *ip*) **double** *real*, **ip*;

modf is the floating-point modulus function. It returns the fractional part of its argument *real*, which is a value *f* in the range 0 <= *f* < 1. It also stores the integral part in the **double** location referenced by *ip*. These numbers satisfy the equation *real* = *f* + **ip*.

Example

This example prompts for a number from the keyboard, then uses **modf** to calculate the number's fractional portion.

```
#include <stdio.h>

main()
{
	extern char *gets();
	extern double modf(), atof();
	double real, fp, ip;
	char string[64];

	for (;;) {
		printf("Enter number: ");
		if (gets(string) == 0)
			break;

		real = atof(string);
		fp = modf(real, &ip);
		printf("%lf is the integral part of %lf\n",
			ip, real);
		printf("%lf is the fractional part of %lf\n",
			fp, real);
	}
}
```

See Also

atof(), ceil(), fabs(), floor(), frexp(), general function, ldexp()

modulus — Definition

Modulus is the operation that returns the remainder of a division operation. For example, 12 modulus four equals zero, because when 12 is divided by four it leaves no remainder. The term "modulo" also refers to the product of a modulus operation; in the above example, the modulo is zero. In C, the modulus operation is indicated with a percent sign '%'; therefore, 12 modulus 4 is written **12%4**.

The modulus operation often is used to trim numbers to a preset range. For example, if you wanted to create a list of single-digit random numbers, you would use the command:

```
rand()%10
```

This is demonstrated by the following example.

Example

This example prints a list of 20 single-digit random numbers. The random-number table is seeded with a portion of the current system time.

```
main()
{
     long nowhere;   /* place to put unused data */
     int counter;

     srand((int)time(&nowhere));
     for (counter = 0; counter <20; counter++)
          printf("%d\n", rand()%10);
}
```

See Also

definitions, operator

Notes

The implementation of C defines how a modulus operator behaves when it operates upon numbers with different signs. On the i8086,

```
10 % -4
```

yields -2. This is not mathematical modulus, which is +2.

mon.h — Header File

Read profile output files
#include <mon.h>

mon.h is used with programs that read the profile output files.

See Also

header files

more — Command

Display text one page at a time
more [**-cdflsu**] [*-window_size*] [*+line_number*] [*+/pattern*] [*file* ...] [-]

more is a filter for paging through text one screenful at a time. *file* is a text file; the operator - tells **more** to read and display the standard input.

Options

more reads options from the command line and from the environmental variable **MORE**. In case of a conflict, the options given on the command line take precedence. Every cluster of options must be preceded with a hyphen '-', even if passed via the environmental variable **MORE**.

more recognizes the following options:

-c Paint the screen from the top line down. **more** normally repaints the screen by scrolling from the bottom of the screen.

-d Prompt the user at the end of each screen with the message:

```
[Press space to continue, 'q' to quit.]
```

The default is to not issue a prompt.

-f Count actual lines from the input file rather than screen lines. This option is useful when the input contains escape sequences that **more** does not recognize.

-l Do not treat the formfeed character **<ctrl-L>** as special. By default, **more** pauses at each formfeed character, as if a full screen had been displayed.

-s Squeeze consecutive blank lines into one blank line. This is useful for looking at **nroff** output, such as manual pages.

-u Display backspaces as control characters and leave the carriage return-linefeed (CR-LF) pair alone. By default, **more** displays backspaces that appear adjacent to an underscore character as underlined text; backspaces that appear between two identical characters as emboldened text; and compresses CR-LF sequences.

+/pattern
Search for *pattern* before displaying a file. *pattern* is a regular expression, as recognized by commands **ed** or **egrep**. *pattern* should be escaped to avoid being processed by the shell.

-window_size
Set the size of the window that **more** displays to *window_size*, which is a decimal integer less than or equal to the number of lines on your terminal. The default window size is read from the **termcap** description for your terminal.

+line_number
Make *line_number* the beginning line to display in *file*. *line_number* is a decimal integer less than the number of lines in *file*.

Commands

The following describes **more**'s interactive commands. These commands are based on those for the UNIX editor **vi**. Some commands may optionally be preceded by a decimal number. If you enter an invalid command, **more** will beep at you.

h
? Help: display a summary of these commands.

*[N]***<space>**
Display the next *N* lines of text (default, one screenful).

*[N]***z**
If *N* is not specified, display the next screenful. Otherwise, display *N* lines and set the default scrolling size to *N* for all subsequent **<space>** and **z** commands.

*[N]***<ctrl-F>**
*[N]***f**
Scroll forward *N* screenfuls (default, one screenful). If *N* is more than the screen size, only the final screenful is displayed.

*[N]***<ctrl-B>**
*[N]***b**
Scroll backward *N* screenfuls (default, one screenful). If *N* is more than the screen size, only the final screenful is displayed.

*[N]***s**
Skip forward *N* lines (default, one line) and display one screenful.

*[N]***<return>**
*[N]***<enter>**
Scroll forward *N* lines (default, one). Display all *N* lines, even if *N* is more than the screen size.

*[N]***<ctrl-D>**
*[N]***d**
Scroll forward *N* lines (default, one half of the screen size). If *N* is specified, it becomes the new default for subsequent **d** and **<ctrl-D>** commands.

<ctrl-L>
Redraw the screen.

' (Apostrophe) Return to the position in the current file where the previous search command started, or to the beginning of the file if no search commands have occurred. This information is lost when a new file is examined.

[N]/pattern
Search forward for the *N*-th line that contains *pattern* (default, one). *pattern* is a regular expression, as recognized by **ed** or **egrep**. The search starts at the second line displayed.

n Repeat previous search.

:f Display the name of the current file with the current line number.

*[N]***:n**
Examine the *N*-th file after the current file, as given in the command line (default, the next file).

*[N]***:p**
Examine the *N*-th file previous to the current file, as given in the command line (default, the previous file).

! *command*
:! *command*
Pass *command* to the shell specified by environment variable **SHELL** for execution. The default shell is **/bin/sh**.

v Invoke an editor to edit the current file. The editor is set by the environment variables **VISUAL** and **EDITOR**, in that order. If these variables are not set, use **vi**.

= Display the current line number.

q

:q
Q
:Q Quit.

Environment

more uses the following environment variables:

EDITOR Specify default editor.

MORE Set default options for **more**

SHELL Specify the shell being used (normally set at login time).

TERM Specify the type of terminal you are using. **more** uses this variable to read from **/etc/termcap** the terminal characteristics needed to manipulate the screen.

VISUAL Specify default visual editor.

See Also

commands, egrep, scat, vi

Author

This software is derived from software contributed to Berkeley by Mark Nudleman. **more** is copyright © 1988,1990 by The Regents of the University of California. Copyright © 1988 by Mark Nudleman. All rights reserved.

more is distributed as a service to COHERENT customers, as is. It is not supported by Mark Williams Company. *Caveat utilitor.*

motd — System Maintenance

File that holds message of the day
/etc/motd

The file **motd** holds the message of the day. Its contents are displayed on every user's screen whenever he logs in.

Only the superuser can alter the contents of this file.

See Also

login, system maintenance

mount() — COHERENT System Call (libc)

Mount a file system
#include <sys/mount.h>
#include <sys/filsys.h>
mount (*special, name,flag*)
char ******special*, ******name*; **int** *flag*;

mount() is the COHERENT system call that mounts a file system. *special* names the physical device that through which the file system is accessed. *name* names the root directory of the newly mounted file system. *flag* controls the manner in which the file system is mounted, as set in header file **sys/mount.h**.

See Also

COHERENT system calls, fd

mount.all — System Maintenance

Mount file systems at boot time
/etc/mount.all

The file **/etc/mount.all** holds a set of **mount** commands to mount all COHERENT file systems on hard disk. It is invoked by the script **/etc/rc**, which COHERENT reads and executes at boot-time.

When you add a new COHERENT partition to your system, you should insert an appropriate entry into this file, to ensure that the new partition is mounted whenever you reboot your system.

See Also

checklist, mount, rc, system maintenance

mount — Command

Mount a file system
/etc/mount [*special directory* [**-ru**]]

mount mounts a file system from the block special file *special* onto *directory* in the system's directory hierarchy. This operation makes the root directory of the mounted file system accessible using the specified *directory* name.

If the **-r** option is specified, the file system is read-only. This is useful for preventing inadvertent changes to precious file systems. The system will not update information such as access times if the **-r** option is used.

The **-u** option causes **mount** to write an entry into the mount table file **/etc/mtab** without actually performing the mount. This is used to note the file system.

When invoked with no arguments, **mount** summarizes the mounted file systems and where they attach.

The command **umount** unmounts a previously mounted file system.

The script **/bin/mount** calls **/etc/mount**, and provides convenient abbreviations for commonly used devices. For example,

```
mount f0
```

executes the command:

```
/etc/mount /dev/fha0 /f0
```

The system administrator should edit this script to reflect the devices used on your system.

Files

/etc/mtab — Mount table
/etc/mnttab — Mount table
/bin/mount — Shell script that calls **/etc/mount**

See Also

commands, fsck, mkfs, mknod, umount

Diagnostics

Errors can occur if *special* or *directory* does not exist or if the user has no permissions on *special*.

The message

```
/etc/mtab older than /etc/boottime
```

indicates that **/etc/mtab** has probably been invalidated by booting the system.

Attempting to **mount** a block-special file which does not contain a COHERENT file system may have disastrous consequences. **mkfs** must be used to create a file system on a blank disk before it is mounted.

mount.h — Header File

Define the mount table
#include <sys/mount.h>

mount.h defines the structures and constants that constitute the COHERENT system's mount table. It also declares functions that are used internally by routines that manipulate the mount table.

See Also

header files

mout() — Multiple-Precision Mathematics

Write multiple-precision integer to stdout
#include <mprec.h>
void mout(*a***)**
mint *a;

The COHERENT system includes a suite of routines that allow you to perform multiple-precision mathematics. The function **mout** writes the multiple-precision integer (or **mint**) pointed to by *a* onto the standard output. The base of the output is set by the value of the external variable **obase**.

See Also

multiple-precision mathematics

mprec.h — Header File

Multiple-precision arithmetic
#include <mprec.h>

The header file **mprec.h** declares a set of routines used to perform multiple-precision arithmetic. It also declares the structure **mint**, which holds multiple-precision integers.

See Also

header files, multiple-precision arithmetic

ms — Technical Information

Manuscript macro package
nroff -ms *file* ...

The **nroff** macro package **ms** formats manuscripts. The tutorial on **nroff** describes the **ms** macros in detail.

ms includes the following macros:

.AB Begin the abstract portion of a document's title page.

.AE End the abstract

.AI Indicate author's institution on a document's title page.

.AU Name the author on the title page of a document.

.B Boldface font: set the following argument in boldface. If the argument is longer than one word, it must be enclosed in quotation marks. Anything on the line after the argument is thrown away.

.BD Block-centered display. Take a portion of text; do not adjust it or break it between two lines, but center it as a whole.

.BT Bottom title. This controls the printing of the footer title, should you want one. It uses three strings, all or any of which can be defined by the user: **LF**, for left-hand portion; **CF**, for center portion; and **RF**, for right-hand portion. **CF** has the default definition of printing the page number; the other two strings are undefined.

.CD Centered display. Center individually every line within a display.

.DA Set the date.

.DE Mark the end of a display. Do *not* use after the macros **.LD**, **.CD**, or **.RD**.

.DS Mark the beginning of a display. Do *not* use for displays longer than one page.

.FE Mark the end of a footnote entry.

.FS Mark the beginning of a footenote entry.

.I Italic font. Used like **.B**, above.

.ID Indent a display 1/2 inch before printing.

.IP Indent a paragraph of text before printing. This macro can take two arguments: argument 1 is used as a **tag** that is printed to the left of the first line of the paragraph; argument 2 indicates how far to indent the paragraph, in characters (the default is five characters, or 1/2 inch).

.KE Indicate the end of a *keep*, or a portion of text that must not be broken between two pages.

.KF Start floating keep.

.KS Indicate the beginning of a *keep*.

.LD Set a display flush left; used with displays that are longer than one page.

.NH Set a numbered heading. This macro takes two arguments. Argument 1 is the text of the heading; if longer than one word, it must be enclosed in quotation marks. Argument 2 is the *depth* of numbering; for example, a '4' here would yield a number of the format "1.1.1.1". No number higher than five is accepted here.

.PP Begin a new paragraph.

.QE Mark the end of a quoted paragraph.

.QP Quoted paragraph. Used like **.IP**, above.

.QS Mark the beginning of quoted text; text is indented by five characters (1/2 inch).

.R Roman font. Used like **.B**, above.

.RE Mark the end of a relative indentation.

.RS Mark the beginning of a relative indentation. A relative indentation is a block of text that is indented five characters (1/2 inch) more than the text before it.

.SH Subheading. One line of space is inserted, and the following line of text is set boldface and centered.

.TA Set tabs, in characters.

.TL Title: format the title entry on the cover page of a document.

Files

/usr/lib/tmac.s

See Also

man, nroff, technical information, troff
Introduction to nroff, Text Processing Language, tutorial

MS-DOS — Technical Information

That other operating system

MS-DOS is the native operating system of the IBM-AT and compatible computers. As such, it needs no introduction to most users. Many customers have asked, however, how MS-DOS and COHERENT compare in terms of their capabilities; and many have also asked for a chart that maps familiar MS-DOS commands to their COHERENT equivalents. This article attempts to fulfill these requests.

MS-DOS vs. COHERENT

MS-DOS differs significantly from COHERENT in practically every aspect of its design. For example, its file system is incompatible with COHERENT; its shell **command.com** differs significantly from COHERENT's suite of shells; the manner in which it loads and executes a program differs completely from COHERENT's.

The most noticeable difference in design, however, is that MS-DOS is a single-user, single-process operating system, whereas COHERENT is a multi-user, multi-tasking operating system.

Single-user means that only one user can use MS-DOS at any given time: whoever sits at the keyboard "owns" the machine and all its facilities. *Multi-user* means, of course, that more than one user can use COHERENT at any given time, via terminals or modems plugged into the computer's serial ports. The number of users who can use your COHERENT system at once is limited only by your computer's speed, available memory, and by the number of serial ports that can be plugged into your computer.

Single-tasking means that MS-DOS can do only one task at a time: it loads a program into memory, runs it to completion, then awaits your request to execute another program. *Multi-tasking* means that COHERENT can execute more than one program at a time.

To grasp how multi-tasking can simplify some work, consider the task of formatting floppy disks. Under MS-DOS, you pop the floppy disk into the drive, invoke the MS-DOS program **format**, answer its queries, then go get a cup of coffee while the machine grinds away. Formatting a box of high-density floppy disks ties up your machine for the better part of an hour, which is largely wasted time for you. Under COHERENT, however, you can format a floppy disk in the *background* — that is, you can tell COHERENT to execute the disk-format program unsupervised, and let you work with another program. For example, if you wish to low-level format a 5.25-inch, high-density floppy disk in drive 0 (that is, drive A), use the following command:

```
/etc/fdformat -v /dev/fha0 &
```

Try it. You'll notice that the COHERENT prompt returns immediately: while COHERENT is formatting your disk for you, you can edit a file, play a video game, dial out to a remote system, or even format a second disk in your machine's B drive (should you have one).

Multi-tasking also means that you can program COHERENT to execute programs untended, even while you are away from your machine. The UUCP system is a good example of this feature. UUCP lets you exchange mail and files with remote systems via modem; once the system is set up, it runs automatically, without requiring that you sit at the keyboard to run it.

This discussion only gives you a taste of the advantages COHERENT enjoys over an obsolete system like MS-DOS. The following documents contain information that MS-DOS users will find helpful:

- The tutorial *Using the COHERENT System* introduces COHERENT to new users. If you are new to COHERENT and have not yet read this tutorial, you should do so before you continue any farther.
- The Lexicon article **dos** describes a COHERENT command that lets you transfer files from MS-DOS file systems to COHERENT, and vice versa. The file systems can reside either on floppy disks or on different partitions of your hard disk.
- The Lexicon articles **floppy disks** and **hard disk** discuss the in's and out's of using mass-storage device with COHERENT. The article **floppy disks** in particular discusses in detail all the steps required to format and manipulate MS-DOS-style floppy disks under COHERENT.
- The Lexicon articles **modem**, **printer**, and **terminal** discussion how to connect these devices to COHERENT, and introduce the set of commands with which you can manipulate them under COHERENT.
- The Lexicon article **execution** describes in detail how COHERENT loads and executes a program. This article is aimed at the technically knowledgeable, but neophytes may find parts of it helpful.
- The Lexicon article **commands** summarizes all commands available under the COHERENT system. This article will help you grasp the scope of COHERENT's suite of commands, and will help you explore them systematically.

COHERENT Equivalents to MS-DOS Commands

The following table lists the most commonly used MS-DOS commands, and gives COHERENT equivalents.

Note that often there is no single COHERENT command that equates to a given MS-DOS command. COHERENT often offers several alternatives, and you can select the one that best suits your needs. Every COHERENT command has its own article in the COHERENT Lexicon; look there first for details on how to use the command.

BACKUP

This command copies a directory's files to a formatted floppy disk to back them up. To do so under COHERENT, use the command:

```
find . -print | cpio -oc > /dev/rfha0
```

Note that **cpio** requires a formatted, defect free floppy disk, however you do not need to create a filesystem on the floppy disk prior to using **cpio**.

Note that if you want COHERENT to prompt you before it backs up a file, use the

command:

```
find . -print | cpio -ocr > /dev/rfha0
```

See the article on the archiving command **cpio** for details on this command — especially important if you expect to retrieve your backed-up files.

Note, too, that the device **/dev/rfha0** corresponds to a 5.25-inch, high-density floppy disk in drive 0 (drive A). See the article **floppy disks** for a list of the devices that correspond to different sizes and configuration of floppy disks.

BREAK

Abort a command. Aborting a command under COHERENT varies, depending upon whether the command is running in the foreground or the background. The keystroke

```
<ctrl-c>
```

aborts most commands that are running in the foreground. To abort a command that is running in the background, you must use the **kill** command. See its Lexicon entry for details on how to use it.

CHDIR or **CD**

Change to another directory. To do so under COHERENT, use the command

cd *dir*

where *dir* is the directory to which you wish to go. The directories '.' and '..' are used by both COHERENT and MS-DOS; since MS-DOS "borrowed" its directory structure from UNIX (of which COHERENT is an implementation), the similarity should not be surprising.

Note that MS-DOS requires that before you can change to directory on another physical device or partition, you must first switch to that device by typing its name before you use the **chdir** command. COHERENT has no such restriction.

CHKDSK

Check the integrity of a file system. Under COHERENT, use the command:

/etc/fsck [*option*] [*filesystem*]

See the Lexicon on **fsck** for details on how to use it.

COMP Compare the contents of two files. To do so under COHERENT, use the following command to compare two binary files:

cmp [*option*] *file1 file2*

cmp displays the bytes which differ between the files.

To compare the contents of two text files, use the command:

diff [*option*] *file1 file2*

COPY Copy the contents of one file into another; create the target file if it does not already exist. Under COHERENT, say:

cp *oldfilename newfilename*

To copy a set of files into a directory without changing their names, use the following form of the command:

```
cp file1 ... fileN directory
```

DATE Reset the current date and time. Under COHERENT, use the command:

```
date yymmddhhmm.ss
```

Only the superuser can reset the system's date and time. When **date** is used without an argument, it prints the date and time on the standard output.

DIR Type the contents of a directory. Under COHERENT, use the command:

```
ls -l
```

DIR/W List a directory's contents in columnar form. Under COHERENT, use either the command:

```
lc
```

or the command:

```
ls -C
```

DISKCOPY
Copy one floppy disk track-by-track to another floppy disk. COHERENT has no exact equivalent to this command; however, you can copy the contents of one disk to another by using the following set of commands.

First, place a write-protect tab on your source disk; insert the disk into drive 0 (drive A), then type the following command:

```
dd if=/dev/fha0 of=/tmp/filename
```

This copies the contents of the 5.25-inch, high-density floppy disk in drive 0 into file **/tmp/filename**. For a table of devices that correspond to other sizes and configurations of floppy disks, see the Lexicon article **floppy disks**.

Second, insert formatted destination diskette into drive 0, and then type the command:

```
dd if=/tmp/filename of=/dev/fha0
```

This command copies the files in directory **/tmp/filename** onto the target floppy disk. Note that the target disk must be formatted before it can receive files; see the Lexicon article **floppy disks** for information on how to do this.

EDLIN Perform simple-minded editing of text files. Under COHERENT, the **ed** editor performs line editing, but is much more sophisticated than **edlin**. COHERENT also includes the **vi** and MicroEMACS screen editors, which are more useful still.

ERASE or **DEL**
Remove a file or a directory. To erase a file, use the command:

```
rm file1 [ ... fileN ]
```

To erase a directory, use the command:

```
rmdir directory
```

To erase a directory and all files and directories below it, use the command:

```
rm -r directory
```

FIND Find a pattern within a text file. Under COHERENT, use the command:

```
grep [option] pattern [file ...]
```

grep is an extremely useful command; see its Lexicon entry for details on how to use it.

MEM Find how much space is left free on your hard disk. Under COHERENT, say:

```
df [options]
```

See the Lexicon entry on **df** for details.

MKDIR Create a new directory. Under COHERENT:

```
mkdir directory ...
```

MODE Set parameters for terminals and ports. Under COHERENT, use the command **stty**. This command comes with many options; see its Lexicon entry for details. The default speeds of all ports and terminals reside in file **/etc/ttys**. The superuser can use a text editor to edit this file to change any or all default settings.

MORE Display text a screenful at a time. Under COHERENT, the command **more** does the same thing, as does the command **scat**.

PRINT Print files via a serial port. To print a file on a dot-matrix printer, use the command:

```
lpr file1 [ ... fileN ]
```

To print a file on a Hewlett-Packard LaserJet printer, use the command

```
hpr file1 [ ... fileN ]
```

Note that before these commands can be used, the appropriate devices must be linked to your system. See the Lexicon article on **printer** for details.

Note, too, that COHERENT uses a spooling system to manage the printing of files; thus, attempting to print a non-existent file will not hang the system.

PROMPT
Change the **command.com** prompt. The COHERENT shells store the prompt format within the environmental variable **PS1**. This variable is usually defined in each user's **.profile** file; this file holds commands that are executed whenever the user logs in. To change the definition of your prompt, edit **.profile** to define **PS1** to suit your preference, then log in again.

Note that the information that can be embedded within the prompt varies between the Bourne and Korn shells. See the Lexicon articles **sh** and **ksh** for details on those shells and their prompts.

RENAME
Rename a file. Under COHERENT, use the command:

```
mv oldfile newfile
```

mv can also be used to move files from one directory or file system to another.

RESTORE

Restore a file saved with the **BACKUP** command. Under COHERENT, insert the floppy disk upon which the **cpio** utility saved its backup archive; then type the command:

```
cpio -icv < /dev/rfha0
```

Note that this command assumes you are using **/dev/rfha0**, which describes a 5.25-inch, high-density floppy disk in drive 0 (drive A). For a table of devices that correspond to other sizes and configurations of floppy disks, see the Lexicon article **floppy disks**.

TREE List all directories on a file system. Under COHERENT, use the command:

```
find / -type d | more
```

To list all files and directories that are subordinate to the current directory, use the command:

```
find . | more
```

The COHERENT command **ls -lR** also lists a directory tree, in a somewhat different output format.

See Also

dos, floppy disks, hard disk, modem, printer, terminal, technical information

msg — Device Driver

Message device driver

The file **/dev/msg** is an interface to the message device driver. It is assigned major device 25 (minor device 0) and can be accessed as a character-special device.

All messaging operations are performed through the COHERENT system call **ioctl**. Each of the operations **msgctl**, **msgget**, **msgsnd**, and **msgrcv** is performed with an integer array as its parameter. The first element of the array is reserved for the return value (default, -1). Subsequent elements represent arguments. The call to **ioctl** passes **MSGCTL**, **MSGGET**, **MSGSND**, or **MSGRCV** as the second argument, and an array of parameters as the third argument. The first argument is an open file descriptor to **/dev/msg**.

Prior to accessing the devices, a entry must be created in directory **/dev**, as follows:

```
/etc/mknod /dev/msg c 25 0
/bin/chmod 444 /dev/msg
```

Notes

The space allocated for message text is set by the kernel variables **NMSG** and **NMSC**. These set, respective, the number of message queues and the number of messages. The total space allocated (**NMSG * NMSC**) must be less than 64 kilobytes.

Allocation of too many message queues or messages can exhaust kernel data space, thus preventing the system from running. Recommended safe limits are **NMSQID**=16 and **NMSG**=100. The values of these variables can be reset using the program **/conf/patch**; you should only do so, however, if you are thoroughly familiar with the workings of the

COHERENT kernel.

Private message queues are not supported. Message queues must be removed manually when no longer required. Queue identifiers consist of a scaled slot number plus a slot usage sequence number. Using the system call **msgctl** with the option **IPC_STAT** will obtain information on the specified slot, even when it returns an error.

To remove all message queues, compile and run the following C code:

```
msgget( 0, 0 );    /* must do first */

for ( qid = 0x100; qid < 0x4000; qid += 0x100 ) {
      struct msqid_ds msb;
      msb.msg_perm.seq = 0;
      msgctl( qid, IPC_STAT, &msb );

      if ( msb.msg_perm.seq > 0 )
            msgctl (msb.msg_perm.seq, IPC_RMID, 0 );
}
```

To load **msg** use the command **drvld**.

Files

/usr/include/sys/ipc.h
/usr/include/sys/msg.h
/dev/msg

See Also

device drivers, drvld, msgctl(), msgget(), msgop()

msg — Command

Send a brief message to other users
msg *user*
message

The command **msg** prints the one-line *message* on the screen of *user*.

The message is send as soon as you type **<return>** on the *message* line. If *user* is not logged in or is not known to the system, **msg** prints an error message on your screen.

See Also

commands

msg.h — Header File

Definitions for message facility
#include <sys/msg.h>

msg.h defines the structures and constants used with the COHERENT message facility.

See Also

header files

msgctl() — COHERENT System Call

Message control operations
#include <sys/msg.h>
int msgctl(*msqid, cmd, buf*)
int *msqid*; **int** *cmd*; **struct msqid_ds** **buf*;

msgctl performs the message-control operations specified by *cmd*. The following *cmd*s are available:

IPC_STAT Place the current value of each member of the data structure associated with *msqid* into the structure pointed to by *buf*.

IPC_SET Set the value of the following members of the data structure associated with *msqid* to the corresponding value found in the structure pointed to by *buf*:

```
msg_perm.uid
msg_perm.gid
msg_perm.mode /* only low 9 bits */
msg_qbytes
```

This *cmd* can only be executed by a process that has an effective-user identifier equal to either that of superuser or to the value of **msg_perm.uid** in the data structure associated with *msqid*. Only superuser can raise the value of **msg_qbytes**.

IPC_RMID Remove the system identifier specified by *msqid* from the system and destroy the message queue and data structure associated with it. This *cmd* can only be executed by a process that has an effective-user identifier equal to either that of superuser or to the value of **msg_perm.uid** in the data structure associated with *msqid*.

msgctl fails if any of the following are true:

- *msqid* is not a valid message queue identifier. **msgctl** sets the global variable **errno** to **EINVAL**.
- *cmd* is not a valid command (**EINVAL**).
- *cmd* is equal to **IPC_STAT** and operation permission is denied to the calling process (**EACCES**).
- *cmd* is equal to **IPC_RMID** or **IPC_SET**, and the effective-user identifier of the calling process is not equal to that of superuser and it is not equal to the value of **msg_perm.uid** in the data structure associated with *msqid* (**EPERM**).
- *cmd* is equal to **IPC_SET**, an attempt is being made to increase to the value of **msg_qbytes**, and the effective-user identifier of the calling process is not equal to that of super user (**EPERM**).
- *buf* points to an illegal address (**EFAULT**).

Return Value

Upon successful completion, **msgctl** returns zero. If a problem occurs, it returns -1 and sets **errno** to an appropriate value.

Files

/usr/include/sys/ipc.h
/usr/include/sys/msg.h
/dev/msg

See Also

COHERENT system calls, msg, msgget(), msgrcv(), msgsnd()

Notes

To improve portability, COHERENT implements the **msg** functions as a device driver rather than as an actual system call.

msgget() — COHERENT System Call

Get message queue

#include <sys/msg.h>
msgget(*key*, *msgflg*)
key_t *key*; **int** *msgflg*;

msgget returns the message queue identifier associated with *key*, should it exist. If *key* has no message queue associated with it, **msgget** checks whether (*msgflg* & **IPC_CREAT**) is true; if it is, then **msgget** creates a message queue identifier and associated message queue and data structure for *key*.

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:

- **msg_perm.cuid**, **msg_perm.uid**, **msg_perm.cgid**, and **msg_perm.gid** are set to, respectively, the effective user identifier and effective group identifier of the calling process.
- The low-order nine bits of **msg_perm.mode** are set to the low-order nine bits of *msgflg*. These nine bits define access permissions: the top three bits specify the owner's access permissions (read, write, execute), the middle three bits specify the owning group's access permissions, and the low three bits specify access permissions for others.
- **msg_ctime** is set to the current time.
- **msg_qbytes** is set equal to the system limit, as defined by the kernel variable NMSQB.

msgget fails if any of the following is true. The term within parentheses gives the value to which **msgget** sets **errno**, as defined in the header file **errno.h**:

- A message queue identifier exists for *key* but operation permission as specified by the low-order nine bits of *msgflg* would not be granted (**EACCES**).
- A message queue identifier does not exist for *key* and (*msgflg* & **IPC_CREAT**) is false (**ENOENT**).
- A message queue identifier is to be created but the number of message queue identifiers system-wide exceeds the system limit as specified in the kernel variable **NMSQID** (**ENOSPC**).
- A message queue identifier exists for *key*, but ((*msgflg* & **IPC_CREAT**) **&&** (*msgflg* & **IPC_EXCL**)) is true (**EEXIST**).

Return Value

Upon successful completion, **msgget** returns the message-queue identifier, which is always a non-negative integer. Otherwise, it returns -1 and sets **errno** to an appropriate value.

Files

/usr/include/sys/ipc.h
/usr/include/sys/msg.h
/dev/msg

See Also

COHERENT system calls, msg, msgctl(), msgrcv(), msgsnd()

Notes

To improve portability, the **msg** functions are presently implemented as a device driver rather than as an actual system call.

msgrcv() — COHERENT System Call

Receive a message
#include <sys/msg.h>
msgrcv(*msqid*, *msgp*, *msgsz*, *msgtyp*, *msgflg*)
int *msqid*, *msgsz*, *msgflg*; **struct msgbuf** **msgp*; **long** *msgtyp*;

msgrcv reads a message from the queue associated with the queue identifier *msqid* and writes it in the structure pointed to by *msgp*. This structure consists of the following members:

```
long mtype;     /* message type */
char mtext[];   /* message text */
```

mtype is the received message's type, as specified by the sending process. *mtext* is the text of the message. *msgsz* gives the size of *mtext*, in bytes. The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and (*msgflg* & **MSG_NOERROR**) is true. The truncated portion of the message is lost, with no indication given to the calling process.

msgtyp specifies the type of message requested, as follows:

- If *msgtyp* equals 0L, the first message on the queue is received.
- If *msgtyp* is greater than 0L, the first message of type *msgtyp* is received.
- If *msgtyp* is less than 0L, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

msgflg specifies the action taken if a message of the desired type is not on the queue, as follows:

- If (*msgflg* & **IPC_NOWAIT**) is true, the calling process immediately returns -1 and sets **errno** to **ENOMSG**.
- If (*msgflg* & **IPC_NOWAIT**) is false, the calling process suspends execution until one of the following occurs:
  1. A message of the desired type is placed on the queue.
  2. *msqid* is removed from the system. When this occurs, **msgrcv** sets **errno** to **EDOM**
  3. The calling process receives a signal that is to be caught. In this case, a message is not received and the calling process resumes execution in the manner prescribed in **signal**.

msgrcv fails and no message is received if any of the following is true:

- *msqid* is not a valid message queue identifier. **msgrcv errno** to **EINVAL**.
- Operation permission is denied to the calling process (**EACCES**).
- *msgsz* is less than zero (**EINVAL**).

- *mtext* is greater than *msgsz* and (*msgflg* & **MSG_NOERROR**) is false (**E2BIG**).
- The queue does not contain a message of the desired type and (*msgtyp* & **IPC_NOWAIT**) is true (**ENOMSG**).
- *msgp* points to an illegal address (**EFAULT**).

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid*:

- **msg_qnum** is decremented by one.
- **msg_lrpid** is set equal to the process identifier of the calling process.
- **msg_rtime** is set equal to the current time.

Return Values

If **msgrcv** returns due to the receipt of a signal, it returns -1 and sets **errno** to **EINTR**. If it returns due to the removal of *msqid* from the system, it returns -1 and sets **errno** to **EDOM**. Upon successful completion, **msgrcv** returns a value equal to the number of bytes written into *mtext*. Otherwise, it returns -1 and sets **errno** to an appropriate value.

Files

/usr/include/sys/ipc.h
/usr/include/sys/msg.h
/dev/msg

See Also

COHERENT system calls, msg, msgctl(), msgget(), msgsnd()

Notes

To improve portability, the COHERENT system implements the **msg** as a device driver rather than as an actual system call.

msgs — Command

Read messages intended for all COHERENT users
msgs [*-q*] [*number*]

msgs selects and displays messages that are intended to be read by all COHERENT users. Messages are mailed to the login **msgs**. They should contain information meant to be read once by most users of the system.

The command **msgs** normally is in a user's **.profile**, so that it is executed every time he logs in. When invoked, it prompts the user with the identifier of the user who sent the message and the message's size. **msgs** then asks the user if he wishes to see the rest of the message. The user should reply with one of the following:

y	Display the message.
<return>	Display the message.
n	Skip this message and go to the next one.
-	Redisplay the last message.
q	Quit **msgs**.
number	Display message *number*; then continue.

If environmental variable **PAGER** is defined, **msgs** will "pipe" each message through the command specified in **PAGER**. For example, the **.profile** command line:

```
export PAGER="exec /bin/scat -1"
```

would invoke **/bin/scat** for each message with the command line argument **-1** (the digit one).

msgs writes into the file **$(HOME)/.msgsrc** the number of the next message the user will see when he invokes **msgs**. **msgs** keeps all messages in the directory **/usr/msgs**; each message is named with a sequential number, which indicates its message number. The file **/usr/msgs/bounds** contains the low and high numbers of the messages in the directory; **msgs** determines whether a user has not read a message by comparing the information in **$(HOME)/.msgsrc** with that in **/usr/msgs/bounds**. If the contents of **/usr/msgs/bounds** are incorrect, the problem can be fixed by removing that file; **msgs** will create a new **bounds** file the next time it is run.

When the contents of a message are no longer needed, simply remove that message. Avoid removing the **bounds** file and the highest numbered message at the same time.

msgs accepts the following command-line options:

-q Query whether there are messages; print "There are new messages" if there are, and "No new messages" if not. The command **msgs -q** is often used in profile scripts.

number Start at message *number* rather than at the message recorded in **$(HOME)/.msgsrc**. If *number* is greater than zero, then start with that message; if *number* is less than zero, then begin *number* messages before the one recorded in **$(HOME)/.msgsrc**.

Files

/usr/spool/mail/msgs — Mail messages file
/usr/msgs/[1-9]* — Data base
/usr/msgs/bounds — File that contains message number bounds
$(HOME)/.msgsrc — Number of next message to be presented

See Also

commands, mail, PAGER, scat

msgsnd() — COHERENT System Call

Send a message
#include <sys/msg.h>
msgsnd(*msqid, msgp, msgsz, msgflg*)
int *msqid, msgsz, msgflg*; **struct msgbuf** **msgp*;

The COHERENT system call **msgsnd** sends a message to the queue associated with the message queue identifier *msqid*. *msgp* points to a structure that contains the message. This structure consists of the following members:

```
long   mtype;     /* message type */
char   mtext[];   /* message text */
```

mtype is a positive long integer that the receiving process uses to select messages. *mtext* is a string that is *msgsz* bytes long. *msgsz* can range from zero to a system-imposed limit as specified in the kernel variable NMSC.

msgflg specifies the action to be taken if one or more of the following are true:

- The number of bytes already on the queue is equal to **msg_qbytes**.
- The number of messages on all queues system-wide equals the system limit specified in the kernel variable NMSG.

msgflg can specify any of the following actions:

- If (*msgflg* & **IPC_NOWAIT**) is true, the message is not sent and the calling process returns immediately.
- If (*msgflg* & **IPC_NOWAIT**) is false, the calling process suspends execution until one of the following occurs:
  1. The condition responsible for the suspension no longer exists, in which case the message is sent.
  2. *msqid* is removed from the system. When this occurs, **msgsnd** sets **errno** to **EDOM** and returns -1.
  3. The calling process receives a signal that is to be caught. In this case, the message is not sent and the calling process resumes execution in the manner prescribed in **signal**.

msgsnd fails and no message is sent if one or more of the following are true:

- *msqid* is not a valid message queue identifier. **msgsnd** sets **errno** to **EINVAL**.
- Operation permission is denied to the calling process (**EACCES**).
- *mtype* is less than one (**EINVAL**).
- The message cannot be sent for one of the reasons cited above and (*msgflg* & **IPC_NOWAIT**) is true (**EAGAIN**).
- *msgsz* is less than zero or greater than the system-imposed limit (**EINVAL**).
- *msgp* points to an illegal address (**EFAULT**).

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid*.

- **msg_qnum** is incremented by one.
- **msg_lspid** is set equal to the process ID of the calling process.
- **msg_stime** is set equal to the current time.

Return Values

If **msgsnd** return because it has received a signal, it returns -1 and sets **errno** to **EINTR**. If it returns because *msqid* was removed from the system, it returns -1 and sets **errno** to **EDOM**.

Upon successful completion, **msgsnd** returns zero. Otherwise, it returns -1 and sets **errno** to an appropriate value.

Files

/usr/include/sys/ipc.h
/usr/include/sys/msg.h
/dev/msg

See Also

COHERENT system calls, msg, msgctl(), msgget(), msgrcv()

Notes

To improve portability, the **msg** functions are presently implemented as a device driver rather than as an actual system call.

msig.h — Header File

Machine-dependent signals
#include <signal.h>

The header file **msig.h** defines the machine-dependent signals that the COHERENT system uses to communicate with its processes. The header file **signal.h** declares constants for the machine-independent signals, and includes **msig.h**.

See Also

header files, signal.h

msqrt() — Multiple-Precision Mathematics

Compute square root of multiple-precision integer
#include <mprec.h>
void msqrt(*a*, *b*, *r*)
mint **a*, **b*, **r*;

The COHERENT system includes a suite of routines that allow you to perform multiple-precision mathematics. The function **msqrt** sets the multiple-precision integer (or **mint**) pointed to by *b* to the integral portion of the positive square root of the **mint** pointed to by *a*. It sets the **mint** pointed to by *r* to the remainder. The value pointed to by *a* must not be negative. The result of the operation is defined by the condition

$$a = b * b + r.$$

See Also

multiple-precision mathematics

msub() — Multiple-Precision Mathematics

Subtract multiple-precision integers
#include <mprec.h>
void msub(*a*, *b*, *c*)
mint **a*, **b*, **c*;

The COHERENT system includes a suite of routines that allow you to perform multiple-precision mathematics. The function **msub** subtracts the multiple-precision integer (or **mint**) pointed to by *a* from the **mint** pointed to by *b*, and writes the result into the **mint** pointed to by *c*.

See Also

multiple-precision mathematics

mtab.h — Header File

Currently mounted file systems
#include <mtab.h>

The file **/etc/mtab** contains an entry for each file system mounted by the **mount** command. This does not include the root file system, which is already mounted when the system boots.

Both the **mount** and **umount** commands use the following structure, defined in **mtab.h**. It contains the name of each special file mounted, the directory upon which it is mounted, and any flags passed to **mount** (such as read only).

```
#define    MNAMSIZ    32
struct     mtab {
     char  mt_name[MNAMSIZ];
     char  mt_special[MNAMSIZ];
     int   mt_flag;
};
```

Files

/etc/mtab
<mtab.h>

See Also

header files, mount, umount

mtioctl.h — Header File

Magnetic-tape I/O control
#include <sys/mtioctl.h>

mtioctl.h defines constants and structures used by routines that control magnetic-tape I/O.

See Also

header files

mtoi() — Multiple-Precision Mathematics

Convert multiple-precision integer to integer
#include <mprec.h>
int mtoi(*a***)**
mint *****a*;

The COHERENT system includes a suite of routines that allow you to perform multiple-precision mathematics. The function **mtoi** returns an integer equal to the value of the multiple-precision integer (or **mint**) pointed to by *a*. The value pointed to by *a* should be in the range allowable for a signed integer.

See Also

multiple-precision mathematics

mtos() — Multiple-Precision Mathematics

Convert multiple-precision integer to string
#include <mprec.h>
char *mtos(*a***) mint *****a***;

The COHERENT system includes a suite of routines that allow you to perform multiple-precision mathematics. The function **mtos** converts the multiple-precision integer (or **mint**) pointed to by *a* to a string. It returns a pointer to the string it creates. The string is allocated by **malloc**, and may be freed by **free**. The base of the string is set by the value of the external variable **obase**.

See Also

multiple-precision mathematics

mtype() — General Function (libc)

Return symbolic machine type
#include <mtype.h>
char *mtype(*type***)**
int *type*;

mtype takes an integer machine *type* and returns the address of a string that contains the symbolic name of the machine. The header file **mtype.h** defines the possible machine types. For example,

```
mtype(M_PDP11)
```

returns the address of the string **PDP-11**.

Files

<mtype.h>

See Also

general functions, l.out.h, ld

Diagnostics

mtype returns NULL to indicate that it doesn't recognize the type of machine requested.

mtype.h — Header File

List processor code numbers
#include <mtype.h>

The header file **mtype.h** assigns a code number to each of the processors supported by Mark Williams C compilers and operating systems. These include the Intel i8086, i8088, i80186, and i80286; the Zilog Z8001 and Z8002; the DEC PDP-11 and VAX; the IBM 370, and the Motorola 68000.

See Also

header file

muit() — Multiple-Precision Mathematics

Multiply multiple-precision integers
#include <mprec.h>
void mult(*a, b, c***)**
mint *a, *b, *c;

The COHERENT system includes a suite of routines that allow you to perform multiple-precision mathematics. The function **mult** multiplies the multiple-precision integers (or **mints**) pointed to by *a* and *b*, and writes the product into the **mint** pointed to by *c*.

See Also

multiple-precision mathematics

multiple-precision mathematics — Overview

The COHERENT system includes the library **libmp**, whose routines allow you to perform multiple precision arithmetic. These functions manipulate a data structure called a **mint**, or "multiple-precision integer," which the header file **mprec.h** defines as follows:

```
typedef struct {
     unsigned len;
     char *val;
} mint;
```

You should not depend on the details of this structure, because on some machines a different representation may be more efficient. Using the listed functions is always safe.

The following gives the multiple-precision routines:

gcd(). . . . . Set variable to greatest common divisor
ispos() . . . . . Return if variable is positive or negative
itom(). . . . . Create a multiple-precision integer
madd() . . . . . Add multiple-precision integers
mcmp(). . . . . Compare multiple-precision integers
mcopy() . . . . . Copy a multiple-precision integer
mdiv(). . . . . Divide multiple-precision integers
min() . . . . . Read multiple-precision integer from stdin
minit() . . . . . Condition global or auto multiple-precision integer
mintfr(). . . . . Free a multiple-precision integer
mitom() . . . . . Reinitialize a multiple-precision integer
mneg() . . . . . Negate multiple-precision integer
mout() . . . . . Write multiple-precision integer to stdout
msqrt() . . . . . Compute square root of multiple-precision integer
msub() . . . . . Subtract multiple-precision integers
mtoi(). . . . . Convert multiple-precision integer to integer
mtos(). . . . . Convert multiple-precision integer to string
mult(). . . . . Multiple multiple-precision integers
mvfree() . . . . . Free multiple-precision integer
pow() . . . . . Raise multiple-precision integer to power
rpow(). . . . . Raise multiple-precision integer to power
sdiv() . . . . . Divide multiple-precision integers
smult() . . . . . Multiply multiple-precision integers
spow(). . . . . Raise multiple-precision integer to power
xgcd(). . . . . Extended greatest-common-divisor function
zerop() . . . . . Indicate if multi-precision integer is zero

itom() creates a new **mint**, initializes it to the signed integer value *n*, and returns a pointer to it. Storage used by a **mint** created with **itom** may be reclaimed using **mintfr()**.

A **mint** that already exists may be reinitialized by **mitom()**, which sets *a* to the value *n*. If the **mint** was declared as a global or automatic variable, it must be conditioned before first use by **minit()**, which prevents garbage values in the **mint** structure from causing chaos. A **mint** conditioned by **minit()** has no value; however, it may be used to receive the result of an operation. For **mints** automatic to a function, **mvfree()** should be used before the function is exited to free the storage used by the **val** field of the **mint** structure. Otherwise, this storage will never be reclaimed.

madd(), **msub()**, and **mult()** set *c* to the sum, difference, or product of *a* and *b*. **mdiv** divides *a* by *b* and writes the quotient and remainder in *q* and *r*. *b* must not be zero. The results of the operation are defined by the following conditions:

1. $a=q*b+r$
2. The sign of *r* equals the sign of *q*

3. The absolute value of *r* < the absolute value of *b*.

smult() is like **mult()**, except the second argument is an integer in the range 0 <= *n* <= 127. **sdiv()** is like **mdiv()**, except the second argument is an integer in the range 1 <= *n* <= 128, and the remainder argument points to an **int** instead of a **mint()**.

pow() sets *c* to *a* raised to the *b* power reduced modulo *m*. **rpow()** sets *c* to *a* raised to the *b* power. **spow()** is like **rpow()**, except the exponent is an integer. In no case may the exponent be negative.

mcopy() sets *b* equal to *a*. **mneg()** sets *b* equal to negative *a*.

msqrt() sets *b* to the integral portion of the positive square root of *a*; *r* is set to the remainder. *a* must not be negative. The result of the operation is defined by the condition

$a = b * b + r$

gcd() sets *c* to the greatest common divisor of *a* and *b*. **xgcd()** is an extended gcd routine that sets *g* to the greatest common divisor of *a* and *b*, and sets *r* and *s* so the relation

$g = a * r + b * s$

holds. For **xgcd()**, *r*, *s* and *g* must all be distinct.

mints may be compared with **mcmp()**, which returns a signed integer less than, equal to, or greater than zero according to whether *a* is less than, equal to, or greater than *b*. **ispos()** returns true (nonzero) if *a* is not negative, false (zero) if *a* is negative. **zerop** returns true if *a* is zero, false otherwise.

mtoi() returns an integer equal to the value of *a*. *a* should be in the allowable range for a signed integer.

The external integers **ibase** and **obase** govern the I/O and ASCII conversion routines. Allowable bases run from two to 16. Permissible digits are 0 through 9 and A through F (lower-case letters are not allowed). **min** reads a **mint** in base **ibase** from the standard input and sets *a* to that value. Leading blanks and an optional leading minus sign are allowed; the number is terminated by the first non-legal digit. **mout()** outputs *a* on the standard output in base **obase. mtos()** performs the same conversion as **mout()**, but the result is placed in a character string instead of being output; a pointer to the string is returned. The string is actually allocated by **malloc()**, and may be freed by **free()**.

mzero() and **mone()** point to **mint**s with values zero and one. **mminint()** and **mmaxint()** point to **mint**s containing the minimum and maximum values that will fit in a signed integer. These constants should never be used as the result of an operation.

All the necessary declarations for these constants and for the library functions are contained in the header file **mprec.h**. They need not be repeated.

To link **mp** modules with an executable object, use the argument **-lmp** with the **cc** or **ld** commands.

Example

The following example converts a string into a multi-precision integer.

```
#include <stdio.h>
#include <mprec.h>
#include <ctype.h>
```

```
/*
 * "ibase" is an int which contains the input base used by "stom".
 * It should be between 2 and 16.
 */
int ibase = 10;

/*
 * stom() reads in a number in base ibase from string 'a' and returns
 * pointer to multiple-precision integer.
 */
mint *stom(s)
register char *s;
{
      char cval;
      mint c = {1, &cval};
      register int ch;
      char mifl = 0;  /* leading minus flag */
      static mint     number;

      mcopy(mzero, &number);     /* set number to zero */
      if ((ch = *s) == '-') {    /* skip leading '-' */
            mifl = 1;
            ch = *++s;
      }

      /* scan thru string 's', building result in "number" */
      while (isascii(ch) && isdigit(ch)) {
            cval = (isdigit(ch) ? ch - '0': ch - 'A');
            smult(&number, ibase, &number);
            madd(&number, &c, &number);
            ch = *++s;
      }

      if (mifl)  /* adjust sign of a "number" */
            mneg(&number, &number);
      return(&number);
}

/* simple test for "stom" */
main()
{
      char  buffer[80];

      printf("Input string ? ");
      gets(buffer);
      mout(stom(buffer)); /* Print in stdout multiple-precision int */
}
```

Files

<mprec.h>
/usr/lib/libmp.a

See Also

bc, dc, libraries, malloc, mprec.h

Diagnostics

On any error, such as division by zero, running out of space or taking the square root of a negative number, an appropriate message is printed on the standard error stream and the program exits with a nonzero status.

mv — Command

Rename files or directories
mv [**-f**] *oldfile* [*newfile*]
mv [**-f**] *file* ... *directory*

mv renames files. In the first form above, it changes the name of *oldfile* to *newfile*. If *newfile* already exists, **mv** replaces it with the file being moved; if not, **mv** creates it. If *newfile* is a directory, **mv** places *oldfile* under that directory.

In the second form, **mv** moves each *file* so that it resides under *directory*. If a file with the new name exists but is unwritable, **mv** will not delete it unless the **-f** option is specified.

mv will not copy directories between devices and will not remove directories that occupy the destination of the command.

Normally, **mv** creates a link to the old file with the new file and then removes the old file. If it cannot create the link (e.g., because the new file is on a different file system than the old), **mv** performs a copy and then removes the old file.

See Also

commands, cp, ln

Notes

mv tests the validity of directory moves by means of search permission. The superuser always has search permission and thus can use **mv** incorrectly.

mvfree() — Multiple-Precision Mathematics

Free multiple-precision integer
#include <mprec.h>
void mvfree(*a*)
mint *****a*;

The COHERENT system includes a suite of routines that allow you to perform multiple-precision mathematics. The function **mvfree** frees the space allocated to an automatic multiple-precision integer (or **mint**). You should call **mvfree** before exiting the function that uses the **mint**, or the storage used by the **val** field of the **mint** structure will never be reclaimed.

See Also

multiple-precision mathematics

N

n.out.h — Header File

Define n.out file structure
#include <n.out.h>

n.out.h defines the **n.out** file structure. This file structure is used to encode executable files; it is the same as the standard COHERENT form **l.out**, except that it uses 32-bit addressing. This file structure is used internally in COHERENT, but is not available under the COHERENT C compiler or assembler.

See Also

header files, l.out

named pipe — Definition

A *named pipe* is a special file created with the command **/etc/mknod**. Unlike the block- and character-special files created by **mknod**, a named pipe is not a device.

A named pipe acts like a conventional pipe set up between related processes. It differs in that it has a visible name that can be seen in a file system. It also differs in that it has permissions (since it's a file and has a name) associated with it just like any other file. This allows a named pipe to be accessed by processes that are *not* related to each other, and can even be used for processes that are running on behalf of difference users.

The following illustrates how one process can write data into a named pipe and an unrelated process can read from it:

```
/etc/mknod my_pipe p            # create the named pipe
chmod 644 my_pipe
ls -lR / > my_pipe &            # pump data into pipe in background
mail fred < my_pipe             # read from the pipe and process
```

This script creates a named pipe called **my_pipe** and makes sure that it is readable; it then pumps a mass of data into the pipe (in the background), and then has a process read data from the named pipe and perform some action on them (in this case, mail the data to use **fred**). In this example, the **mail** process could be running from some other login and could either be in the foreground or background.

See Also

definitions, mknod, pipe

ncheck — Command

Print file names corresponding to i-node
ncheck [**-i** *number* ...] [**-as**] *filesystem* ...

An *i-number* identifies an i-node. **ncheck** generates a list of file names by i-number for each *filesystem*, which should be the name of a device special file that contains a proper COHERENT file system. Using the raw device generally decreases the time **ncheck** requires to do its work.

The output is in the unsorted traversal order of the file system hierarchy. **ncheck** distinguishes directories from files by suffixing '/.' to directory names.

Under the **-i** option, **ncheck** prints the file name corresponding to each i-number *number* in the given list. Under the **-a** option, **ncheck** prints only the names of special files and set user-ID mode files; this option allows the system administrator to ascertain quickly whether

these files represent possible security breaches.

See Also

commands, i-node

Diagnostics

ncheck appends '??' to the generated file name if it cannot find the proper parent structure while retrieving the file-name information. It represents any loops detected in the file name by the characters '...'. Extremely addled file systems may generate other reasonably self-explanatory diagnostics.

newgrp — Command

Change to a new group
newgrp *group*

newgrp changes the user's group identification to the specified *group*, if access is permitted. The file **/etc/group** determines group access. Group access may be unrestricted, or open to all users with specific exceptions, or restricted to certain users via a password.

The shell executes **newgrp** directly.

Files

/etc/group — Give group access

See Also

commands, group, ksh, sh

Diagnostics

If **newgrp** succeeds, no diagnostic is printed.

Notes

Interruption of **newgrp** can result in the user being logged off.

Under the Korn shell, **newgrp** is an alias for **exec newgrp**.

newusr — Command

Add new user to COHERENT system
/etc/newusr *login "User Name" parentdir* [*shell*]

newusr adds *login* to the user base of the COHERENT system. It adds an entry for *login* to the password file **/etc/passwd**. *parentdir* is the parent of the directory *parentdir/login* which is created as the home directory for the new user. *User Name* is the name of the human for whom *login* is being created. *shell* names the shell to be used; the default is the Bourne shell **/bin/sh**.

For example, the command

```
/etc/newusr fwb "Fred Butzen" /v /usr/bin/ksh
```

creates new user Fred Butzen, with login **fwb**, home directory **/v/fwb**, and the Korn shell **/usr/bin/ksh**.

Files

/etc/group — User groups
/etc/passwd — User passwords
/parentdir/user — User home directory
/usr/spool/mail/*user* — User mailbox

See Also

commands, passwd

Diagnostics

newusr complains if an entry for *user* already exists in the password file.

Notes

Only the superuser can add new users to the system with **newusr**.

nkb — Device Driver

Device driver for console keyboard

The COHERENT device-driver **nkb** supports industry-standard 83-, 101-, and 102-key AT-protocol keyboards attached as the computer console.

nkb lets you define both the layout of the keyboard and the values returned by function keys. You can change layout and function-key bindings by using the special keyboard mapping programs kept in directory **/conf/kbd**. This directory contains the C source code for the mapping tables, as well as a **Makefile** that helps you rebuild the mapping programs. See the Lexicon article **keyboard tables** for details.

nkb understands the following "shift" and "lock" keys:

scroll	Scroll lock
num	Keypad NUM lock
caps	Shift or CAPS lock
lalt	Left ALT key
ralt	Right ALT key
lshift	Left SHIFT key
rshift	Right SHIFT key
lctrl	Left CTRL key
rctrl	Right CTRL key
altgr	ALT Graphic key (non-US keyboards)

nkb records an internal shift state, as defined by the current positions of the shift and lock keys. The shift state is a logical combination of internal states **SHIFT**, **CTRL**, **ALT**, and **ALT_GR**. The **lshift** and **rshift** keys combine to form the current **SHIFT** state for non-alphabetic keys. Alphabetic keys generally use the current state of the **caps** lock key in addition to **lshift** and **rshift**. Numeric keys found on the keypad generally use the state of the **num** lock key combined with **lshift** and **rshift**. The two "control" keys, **lctrl** and **rctrl**, form the internal **CTRL** state. In a similar manner, the two "alt" keys, **lalt** and **ralt**, form the internal **ALT** state. Note that 102-key keyboards generally replace the **ralt** key with the **altgr** key, to allow access to the alternate graphics characters found on some keyboards.

nkb lets you configure or read the internal mapping tables via the following **ioctl()** requests, as defined in header file **<sgtty.h>**:

TIOCGETF	Get function key bindings
TIOCSETF	Set function key bindings
TIOCGETKBT	Get keyboard table bindings
TIOCSETKBT	Set keyboard table bindings

Requests **TIOCGETF** and **TIOCSETF** reference a data structure of type **FNKEY**, which is a **typedef** defined in header file **<sys/kb.h>**. Structure member **k_fnval** is a character array that contains a series of contiguous function key/value bindings; the end of the bindings is marked by manifest constant **DELIM**. You can use any value other than **DELIM** as part of a

function-key binding. Structure member **k_nfkeys** indicates how many function keys have associated entries in **k_fnval**. Function keys are numbered from zero through **k_nfkeys**-1.

By convention, function-key 0, when enabled, causes the computer system to reboot. This function key is usually bound to the key sequence **<ctrl><alt><del>**, but you can disable it by setting the value of driver-variable **KBBOOT_** to zero. To do so, use the following command:

```
/conf/patch /coherent KBBOOT_=0
```

Requests **TIOCGETKBT** and **TIOCSETKBT** reference an array that contains **MAX_KEYS** occurrences of data structure **KBTBL**, which is a **typedef** defined in header file **<sys/kb.h>**. Structure member **k_key** contains the *scan code set three* code value for the desired key. Header file **<sys/kbscan.h>** contains manifest (symbolic) constants of the form **K_***nnn*, which map AT keyboard *physical* key number *nnn* to the corresponding scan-code set-three value generated by the keyboard. Note that the **nkb** driver disables the scan-code translation that the keyboard controller normally performs, as well as setting the keyboard to scan code set three.

Structure member **k_val** is a nine-element array that contains the key mappings that correspond to the following index values and shift states:

0	**BASE**
1	**SHIFT**
2	**CTRL**
3	**CTRL_SHIFT**
4	**ALT**
5	**ALT_SHIFT**
6	**ALT_CTRL**
7	**ALT_CTRL_SHIFT**
8	**ALT_GR**

Structure member **k_flags** contains mode information for the given key. One field in **k_flags** indicates the *class* of key. This sub-field lets you specify whether a key is a "shift" key (as defined above), a special or programmable "function" key, or a "regular" key. The following symbolic constants specify the *class* of key:

S The specified key is a "shift" or "lock" key. Note that all entries in array **k_val** must be identical for a "shift" or "lock" key to work correctly.

F The specified key is a "function" or special key. The value of all elements of array **k_val** must specify a function key number. See header file **<kb.h>** for a list of predefined function keys.

O The specified key is "regular" and requires no special processing.

The next sub-field of **k_flags** specifies the *type* of key, as specified in the AT keyboard technical reference. The *type* sub-field specifies under what conditions a given key will generate an interrupt. The possible choices are:

M Make: generate an interrupt only upon key "make" (i.e., when the key is depressed). This mode is useful for keys which do not repeat. Note that using this mode with "shift" keys stops you from unshifting upon release of the key!

T Typematic: generate an interrupt when the key is depressed, and generate subsequent key-depression interrupts while the key is depressed. The rate at which interrupts are generated is specified by the typematic rate of the

keyboard. This type is usually associated with a "regular" key.

MB Make/Break: generate an interrupt when the key is depressed, and when it is released. No additional interrupts are generated no matter how long the key is depressed. This mode is used for "shift" keys.

TMB Typematic/Make/Break: generate an interrupt when the key is first depressed; generate subsequent key depression interrupts while the key remains depressed; and generate an interrupt when the key is released.

The last sub-field of **k_flags** specifies the *lock* keys, if any, that affect the specified key:

C The **caps** lock key that affects this key. If the specified key is depressed while **caps** lock is active, it is equivalent to having used either of the SHIFT keys with this key. When **caps** lock is in effect, use of either of the SHIFT keys temporarily toggles the state of the **caps** lock.

N The **num** lock key affects this key. If the specified key is depressed while **num** lock is active, it is equivalent to having used either of the SHIFT keys in conjunction with the specified key. When **num** lock is in effect, use of either of the SHIFT keys temporarily toggles the state of the **num** lock.

See Also

device drivers, fnkey, keyboard tables

Technical Reference for the IBM Personal Computer AT, IBM Corporation, 1984.

Multi-Function Keyboards: Layouts, Cherry Electrical Products Corp.

nlist() — General Function (libc)

Symbol table lookup
#include <l.out.h>
int nlist(*file, nlp***)**
char ****file*;**
struct nlist ****nlp*;**

nlist searches the name list (symbol table) of the load module *file* for each symbol in the array pointed to by *nlp*. For example, the command **ps** uses this routine on the system load module (**/coherent**) to obtain the addresses of system tables in memory (**/dev/mem**).

nlp points to an array of **nlist** structures, terminated by a structure with a null string as its **n_name** member. The header file **l.out.h** defines **nlist** as follows:

```
#define       NCPLN16

struct nlist {
       char   n_name[NCPLN];
       int    n_type;
       unsignedn_value;
};
```

The caller should set the entry **n_name**; **nlist** will fill in the other entries. **nlist** sets both **n_type** and **n_value** to zero if the symbol is not found.

Files

l.out.h

See Also

general functions, l.out.h, nm, strip

Diagnostics

If *file* is not a load module or has had its symbol table stripped, all returned **n_type** and **n_value** entries will be zero.

nm — Command

Print a program's symbol table
nm [**-adgnopru**] *file* ...

The command **nm** prints the symbol table of each *file*. It can read binary files produced by the compiler, assembler, or linker.

When a C source file is compiled with the **-c** switch to the **cc** command, or when a file of assembly language is assembled, the result is an object module, which is signified by the suffix **.o**.

The linker **ld** links multiple object modules to form an executable program. Frequently used object modules often are grouped by the archiver **ar** into a *library*, which is signified by the suffix **.a**. **nm** can read all three kinds of files: .o, .a, and fully linked executables.

Options

nm recognizes the following options:

-a Print all symbols. Normally, **nm** prints names that are in C-style format and ignores symbols with names inaccessible from C programs.

-d Print only defined symbol.

-g Print only global symbols.

-n Sort numerically rather than alphabetically. **nm** uses unsigned compares when sorting symbols with this option.

-o Append the file name to the beginning of each output line.

-p Print symbols in the order in which they appear within the symbol table.

-r Sort in reverse-alphabetical order.

-u Print only undefined symbols.

Output

Output of **nm** is a series of lines of the form:

addr type symbol

For example,

```
0020 SI main
```

If the input file is a library, symbols are listed separately for each member of the library, preceded by a header line that names the library element. For example, if **foo.a** contains elements **foo1.o** and **foo2.o**, then the output of **nm** is something like this:

```
#nm foo.a
foo1.o:
      U ex1
0002  C ex2
0024 PD ex3
0000 SI fn1
...
foo2.o:
0022 PD ex1
      U ex2
      U fn1
0015 SI fn3
0000 SI main
...
```

The *addr* field gives the value of the symbol in hexadecimal. If the symbol belongs to the instruction or data segment of a program, then the value of the symbol is the offset within that segment. If the value is unknown, this field is left blank.

The *type* field is one of the following (symbol types **PI** and **SD** are relatively obscure and are available only through the assembler):

SI	Shared instruction
PI	Private instruction
BI	Uninitialized instruction
SD	Shared data
PD	Private data
BD	Uninitialized data
D	Debug tables
A	Absolute symbol
C	Reference
U	Undefined

Please note that the *type* field is printed in lower case (e.g., **si** instead of **SI**) if the symbol is local rather than global. By default, the C compiler strips local symbols from the object modules files it creates.

See Also

cc, commands, ld, size, strip

notmem() — General Function (libc)

Check if memory is allocated
int notmem(*ptr***);**
char ****ptr*;**

notmem checks if a memory block has been allocated by **calloc**, **malloc**, or **realloc**. *ptr* points to the block to be checked.

notmem searches the arena for *ptr*. It returns one if *ptr* is not a memory block obtained from **malloc**, **calloc**, or **realloc**, and zero if it is.

Example

The following example prints a string, and frees it if it was **malloc**'d.

```
#include <sys/malloc.h>

pfree(s)
char *s;
{
	printf("%s\n", s);
	if(!notmem(s))
		free(s);
}

main()
{
	char *mallocked_string;
	char notmallocked_string[50];

	if ((mallocked_string = malloc(50)) == NULL)
		exit(1);

	strcpy(mallocked_string, "This is a malloc'd string");
	strcpy(notmallocked_string, "This is not a malloc'd string");

	pfree(mallocked_string);
	pfree(notmallocked_string);
}
```

See Also

arena, calloc(), free(), general functions, malloc(), realloc(), setbuf()

nptx — Command

Generate permutations of users' full names
nptx

The command **nptx** reads an address/name pair (that is, an address and a user's full name), and prints on the standard output as many permutations of the user's name as it can think of, each linked to the given address. A set of such permutations helps to relieve a user of needing to know the exact form of another user's name when she wishes to send mail to that user.

When a set of users' names are filtered through **nptx**, its output can be used as a "full-name data base" that can be used by the COHERENT mail system.

The format of an input line is:

```
address	name
```

address can contain any address. It is terminated by a **<tab>** character. *name* consists of white-space-separated names or initials, with an optional nickname given in parentheses, terminated by either a newline character or a comma.

nptx prints all permutations of the first names and initials, with the last name appearing in each permutation. Permutations are not necesarily unique.

Example

Given the address/name pair

```
chicago!widget!lc     LaMonte Cranston(Shadow)
```

nptx produces the following set of permutations:

```
Cranston              chicago!widget!lc
L.Cranston            chicago!widget!lc
LaMonte.Cranston      chicago!widget!lc
S.Cranston            chicago!widget!lc
Shadow.Cranston       chicago!widget!lc
```

See Also

commands, mail, mkfnames, paths

Notes

The command **mkfnames** can read a file of names and massage them into the form expected by **nptx**.

nptx assumes European-style names, i.e., that the family name comes last (unlike Oriental names, in which the family name comes first).

nroff — Command

Text-formatting language
nroff [*option* ...] [*file* ...]

nroff is the COHERENT text-formatter and text-formatting language. By embedding commands within files of text, you can instruct **nroff** to format text, create paragraphs, subheadings, headers, footers, and in general perform all tasks required to format text for the printed page or for screen display.

nroff is designed to be used with character-display terminals or monospace printers. The related program **troff** performs typeset-quality formatting, suitable for printing on the Hewlett-Packard LaserJet printer or any printer for which the PostScript language has been implemented. **troff**'s formatting language is a superset of that used by **nroff**. Text that you have encoded for formatting by **nroff** will work with **troff**, but the reverse is not always true. See the Lexicon entry on **troff** for information that applies to **troff** alone.

nroff Input

nroff processes each *file*, or the standard input if none is specified, and prints the formatted result on the standard output. The input must contain formatting instructions as well as the text to be processed.

Basic **nroff** commands provide for such things as setting line length, page length, and page offset, generating vertical and horizontal motions, indentation, filling and adjusting output lines, and centering. The great flexibility of **nroff** lies in its acceptance of user-defined macros to control almost all formatting. For example, the formation of paragraphs, header and footer areas, and footnotes must all be implemented by the user via macros.

The following summarizes the commands and options that can be used with **nroff**. Four types of commands and options are described: (1) command line options; (2) **nroff**'s basic commands (also called *primitives*); (3) escape sequences that can be used with **nroff**; and (4) **nroff**'s dedicated number registers, and what information each one keeps.

Command-line Options

Command-line options may be listed in any order on the command line. They are as follows:

-d Debug: print each request before execution. This options is extremely useful when you are writing new macros.

-f *name*
Write the temporary file in file *name*.

-k Keep: do not erase the temporary file.

-i Read from the standard input after reading the given *files*.

-m*name*
Include the macro file **/usr/lib/tmac.***name* in the input stream.

-n*N* Number the first page of output *N*.

-r*aN* Set number register *a* to the value *N*.

-r*abN* Set number register *ab* to value *N*. For obvious reasons, *ab* cannot contain a digit.

-x Do not eject to the bottom of the last page when text ends. Use this option when you wish to use **nroff** interactively. It, too, is useful when debugging macros.

If the environmental variable **NROFF** is set when **nroff** is invoked, its contents are prefixed to list of command-line arguments. This let you set commonly used options once in the environment, rather than having to retype them for each invocation of **nroff**.

Primitives

The following gives the basic commands, or *primitives*, that are built into **nroff**. These primitives can be assembled into macros, or can be written directly into the text of your document. Commands may begin either with a period '.' or with an apostrophe; the former causes a break (see **.br**, below), the latter does not.

.ab *msg*
Abort: print *msg* on the standard error and abort processing.

.ad [bclr]
Enter adjust mode: that is, insert white space between words to create right-justified output. **b** adjusts for both margins; this is the default. **c** adjusts and centers on the line. **l** adjusts, flush with the left margin. **r** adjusts, flush with the right margin.

.af *R X*
Assign format *X* to number register *R*. The assigned format may be one of the following:

1 Arabic numerals (default)
i Lower-case Roman numerals
I Upper-case Roman numerals
a Lower-case alphabetic characters
A Upper-case alphabetic characters

.am *XX*
Append the following to macro *XX*. Used like **.de**, below.

.as *XX* Append the following to string *XX*. Used like **.ds**, below.

.bp Begin a new page.

.br Break; print any fraction of a line of text that is in the input buffer before reading new text.

.c2 *c* Set the no-break control character to *c*. With no argument, reset it to the default apostrophe.

.cc *c* Set the normal control character to *c*. With no argument, reset it to the default period.

.ce *N* Center *N* lines of text (default, one).

.ch *XX N*
Change the location of the trap for macro *XX* to vertical position *N* on the page. Used like command **.wh**, below.

.co *endmark*
Copy input directly to the output until *endmark* is seen. If no *endmark* is given, copy until another **.co** is seen.

.cu Underline continuously.

.da *X* Divert and append the following text into macro *X*. A diversion is ended by a **.da** command that has no argument.

.de *X* Define macro *X*. The macro definition is ended by a line that contains only two periods "..".

.di *X* Divert the following text into macro *X*. Diversion is ended by a **.di** command that has no argument.

.ds *X value*
Define string *X* to have the given *value*.

.ec *c* Set the escape characer to *c*. With no argument, reset it to the default backslash character '\'.

.el *action*
Execute *action* when the test in an **.ie** command fails. This command must be used with an **.ie** command.

.em *XX*
Execute macro *XX* when processing is completed.

.eo Escape off: turn off special handling of all escape sequences.

.ev *N* Change the environment. When followed by 0, 1, 2, the command *pushes* that environment; when used without an argument, the command *pops* the present environment and returns to the previous environment.

.ex Exit from **nroff** without further ado.

.fi Enter fill mode.

.fl Flush; same as **.br**.

.ft *X* Change the current font to *X*. **nroff** recognizes **R**, **B**, and **I**, for Roman, bold, and italic, respectively.

.ie *condition action*
This command tests to see if *condition* is true; if true, it then executes *action*; otherwise, it performs the action introduced by an **.el** primitive. This command must be used with the **.el** command.

.if *condition action*
This command tests to see if *condition* is true; if so, then *action* is executed; otherwise, *action* is ignored. The command **.if o** applies if the page number is odd, and the command **.if e** applies if the page number is even. The command **.if n** applies if the text is processed by **nroff**, and the command **.if t** applies if the text is processed by **troff**. The command **.if l** applies in landscape mode. The command **.if p** applies to **troff** PostScript mode. Note that the last two conditions are unique to the COHERENT implementation of **nroff**, and may not be portable to other implementations.

.ig *X* Ignore all input until macro *.X* is called; if no argument is given, ignore input until two periods "..".

.in *NX*
Change the normal indentation to *N* units of *X* scale. *X* can be **u** or **i**, for *machine units* or *inches*, respectively. If *N* is used without *X*, **nroff** assumes the indentation to be given in number of character-widths (in picas, or tenths of an inch). Default indentation is zero.

.it *N XX*
Set an input trap to execute macro *XX* after *N* input lines (not counting request lines).

.lc *c* Set the leader dot character to *c*. When **nroff** sees the escape sequence **\a**, it fills space to the next tab stop with the leader dot character. **lc** with no argument tells **nroff** to use spaces to fill leaders.

.ll *NX* Set the line length. Used like the **.in** command, above.

.ls *X* Leave spaces; insert *X* vertical spaces after each line of text. Default is zero.

.lt *NX* Length of title. Used like the **.in** command, above.

.na Enter no-adjust mode. Line lengths are not changed.

.ne *NX*
Confirm that at least *N* portions of *X* units of measure of vertical space are needed before the next trap. If this amount of space is not available, then move the text to the top of the next page. *X* can be **i** or **v**, for inches or vertical spaces, respectively. This command is used in display macros and in paragraph macros to help prevent widows and orphans.

.nf Enter no-fill mode; no right justification is performed, although line lengths are changed to approximate uniform line length.

.nh Turn off hyphenation. **nroff** hyphenates according to built-in algorithms that are correct most of the time, but not always.

.nr *X N1 N2*
Set number register *X* to value *N1*; set its default increment/decrement to *N2*. For example, **.nr X 2 3** sets number register **X** to 2, and sets its default increment to 3.

.ns No-space mode.

.nx *file* Terminate processing of the current input file and begin processing *file* instead.

.pl *NX*
Set the page length to *N*. The unit of measure *X* can be **V** or **i**, for vertical spaces (sixths of an inch) or inches, respectively. The default unit of measure is vertical spaces.

.pn *N* Set the page number to *N*.

.po *NX*
Set the default page offset to *N*. The unit of measure *X* can be set to **i**, for inches. The default unit of measure is number of characters.

.rb *file* Read binary: read the given *file* and copy it directly to the output without processing.

.rd *prompt*
Read an insertion from the standard input after issuing the given *prompt*.

.rf *XX YY*
Rename font *XX* as *YY*.

.rm *XX* Remove macro or string *XX*.

.rn *XX YY*
Change the name of a macro or string from *XX* to *YY*.

.rr *X* Remove register *X*.

.rs Restore normal space mode.

.so *file* Open *file*, read its contents, and process them. When the end of *file* is reached, resume processing the contents of the present file.

.sp [|]*NX*
Space down *N*. The unit of measure *X* can be **i**, for inches, with the default unit of measure being vertical spaces, or sixths of an inch. The optional vertical bar '|' indicates that *N* is an absolute value; for example, **.sp |1.5i** means to move to 1.5 inches below the top of the page, whereas **.sp 1.5i** means to move to 1.5 inches below the present position.

.ta *NX* ...
Set the tab to *N*. The unit of measure *X* can be set to **i**, for inches; the default unit of measure is number of characters, or tenths of an inch. A tab setting, of course, is for an absolute, not a relative, value. If more than one tab setting is defined, the first defines the first tabulation character on a text line, the second defines the second tabulation character, etc. Any undefined tabulations are thrown away.

.tc *X N*
Fill any unused space within a tabulation field with the character *X*. If the optional *N* is present, it specifies a width for the character; for example, **.tc . .1i** fills tabs with dots spaced one-tenth of an inch apart.

.ti *NX* Temporary indent; indent only the next line. Used like the **.in** command, above.

.tl *'left'center'right'*
Set a three-part title, with *left* being set flush left, *center* being centered on the line, and *right* being set flush right. Note the use of the apostrophes to separate the fields; the apostrophes for an undefined field must still be present, or a syntax error will be generated.

.tm *message*
Print *message* on the standard error device. This is often used with **.if** or **.ie** commands to indicate an error condition.

.tr *xy* Translate character *x* to *y* on output.

.ul *N* Underline the next *N* lines.

.vs *Np* Reset the normal vertical spacing to *N* points **p**. One point equals 1/72 of an inch; the default setting is 12 points, or 1/6 of an inch.

.wh *NX action*
Set a trap to perform *action* when point *N* is reached on every formatted page. If *N* is negative, it is measured up from the bottom of the page. The unit of measure *X* may be **i** or **v**, for inches or number of vertical lines, respectively; the default unit of measure is **v**.

Escape Sequences

The following lists **nroff**'s escape sequences, or commands that suspend or work around the normal operation of **nroff**. All escape sequences are introduced by the *escape character*, normally the backslash character '\'.

\(*xx* Print special character *xx*, as defined by a **.dc** request. **nroff** reads default special character definitions from file **/usr/lib/roff/nroff/specials.r**. For example, the escape sequence **\(<=** prints the less-than-or-equal-to symbol ≤.

**** Print a backslash character. This can be used to print a literal backslash character in the output text, or to defer the interpretation of a macro or string from the time it is processed to the time that it is called.

\- Print a minus sign.

\& Ignore what is normally a command string.

\$*N* Call macro argument *N*.

\" Introduce a comment within your text. All text to the right of this escape sequence will be ignored by **nroff**. This sequence must read **.\"** when used at the beginning of a line.

*****S* Call string *S*.

(ST*
Call string *ST*.

\a Fill the space to the next tab stop with leader dots (normally '.').

\e Print the escape character in the output text.

\f*X* Set font to *X*; this can be either **R**, **I**, **B**, or **P**, for Roman, *italic*, **bold**, or previous font, respectively.

\h'[|]*NX***'**
Move horizontally by *N* units of *X*. If *N* is positive, move to the right; if negative, move to the left. The unit of measure *X* may be **i**, for inches; the default unit of measure is character-widths. When the optional vertical bar '|' is used, move to an absolute position on the line. For example **\h'|1.5i'** moves to 1.5 inches to the right of the left margin, whereas **\h'1.5i'** moves 1.5 inches to the right of the current position.

\l'*NX***'** Draw a horizontal line *N* units of *X* long. The unit of measure *X* may be **i**, for inches; the default unit of measure is character-widths.

\L'*NX***'** Draw a vertical line; used like **\l**, above.

\n*X* Read the value of number register *X*.

\n(*XY* Read the value of number register *XY*.

\o'*chars***'**
Overstrike the given *chars*, centered on the widest.

\t Print a tab.

\v'*NX***'** Vertical motion; move *N* units of *X* vertically. If *N* is positive, move down; if negative, move up. The unit of measure *X* may be **i** or **v**, for inches or vertical spaces (sixths of an inch), respectively. The default unit of measure is **v**.

\w'*argument***'**
Measure the width of *argument*. For example

```
\w'stuff and nonsense'
```

measures the width of the phrase **stuff and nonsense**; or

```
\w'\$1'
```

measures the width of the first argument passed to a macro, whatever that argument might happen to be. Therefore, the command **.in \w'\$1'** will indent a line by the width of argument 1.

\X*dd* Output the character with hexadecimal value *dd*, where *dd* are two hexadecimal digits.

\z*c* Print character *c* with zero width.

\<newline>
Ignore this **<newline>** character.

\{ Begin conditional commands; used after an **.if**, an **.ie**, or an **.el** command.

\{ Begin conditional commands, and ignore the following carriage return.

\} End conditional commands.

Dedicated Number Registers

The following lists the number registers that are predefined in **nroff**. You can read or reset these registers to suit the need of any special formats that you wish to devise.

.$ Number of arguments in a call to a macro.

% Present page number.

dy Day of the month, as set by COHERENT.

.i Present level of indentation.

.l Present line length.

mo Month, as set by COHERENT.

.o Present page offset.

.p Page length.

yr Year, as set by COHERENT.

Printer Configuration

nroff reads several files in directory **/usr/lib/roff/nroff** to find printer-specific information. It reads special character definitions from file **specials.r.** If file **fonts.r** exists, **nroff** reads font information from it; **nroff** understands only Roman, bold and italic fonts, but **.rf** requests may define alternative font names. If file **.pre** exists, **nroff** copies it at the beginning of the output. If file **.post** exists, **nroff** copies it at the end of the output. In landscape mode, **nroff** looks for files **.pre_land** and **.post_land** instead. You can change these files as desired to include printer-specific commands in **nroff** output.

Miscellaneous

The **-ms** macro package is kept in file **/usr/lib/tmac.s.** The macros in this package are more than sufficient for most ordinary text processing. Beginners should work through this macro package rather than trying to deal at once with the basic program.

The tutorial to **nroff**, which is included with this manual, provides a detailed introduction to **nroff**. Error messages for **nroff** appear in the appendix to this manual.

Files

/tmp/rof* — Temporary files
/usr/lib/tmac.* — Standard macro packages
/usr/lib/roff/nroff/ — Support files directory
/usr/lib/roff/nroff/.pre — Output prefix
/usr/lib/roff/nroff/.pre_land — Output prefix, landscape mode
/usr/lib/roff/nroff/.post — Output suffix
/usr/lib/roff/nroff/.post_land — Output suffix, landscape mode
/usr/lib/roff/nroff/fonts.r — Alternative font name definitions
/usr/lib/roff/nroff/specials.r — Special character definitions

See Also

col, commands, deroff, man, ms, troff
nroff, The Text-Formatting Language, tutorial

Notes

You should avoid using characters with values 0x01 through 0x1F, and those with values 0x80 through 0x9F. These are reserved for internal use by **nroff** and **troff**, and using them in your input will cause errors.

NUL — Definition

NUL is the ASCII null character '\0'. Do not confuse it with the null pointer NULL or with the empty string "". A C-language string is always terminated with a NUL. The empty string "" is an array of **char**s with only one element, namely a NUL.

See Also

ASCII, definitions, NULL

NULL — Definition

NULL is defined in the header file **stddef.h**. It is the null pointer **(char *)0**, which is a pointer initialized to zero. Numerous routines return this value to indicate failure; it is useful as a return value because it points nowhere, and so removes the possibility of accidentally destroying a section of memory after failure.

See Also

definitions, NUL, pointer, stdio.h

null — Device Driver

The "bit bucket"

All data written to the special file **/dev/null** is thrown away (sent to the "bit bucket"). This is useful, for example, to test a program's side effects while ignoring its output.

A read from file **/dev/null** returns end of file (zero bytes of data). The shell **sh** uses **/dev/null** as input to background processes.

Files

/dev/null

See Also

device drivers, sh

nybble — Definition

A **nybble** is four bits, or half of an eight-bit byte. The term is generally used to refer to the low four bits or the high four bits of a byte. Thus, a byte may be said to have a "low nybble" and a "high nybble". One nybble encodes one hexadecimal digit.

See Also

bit, byte, definitions

O

object format — Definition

An **object format** describes the form of compiled program that still contains relocation information. The linker **ld** reads file in object format to create executable files.

COHERENT creates object modules that are in the format **l.out**.

See Also

definitions, l.out, ld

od — Command

Print an octal dump of a file
od [-bcdox] [*file*] [[+] *offset*[.][b]]

od prints the specified *file* as a sequence of octal numbers, or machine words. If no *file* is specified, **od** dumps the standard input.

The following options set the format of **od**'s output:

-b	Bytes in default base
-c	Bytes in ASCII characters
-d	Words in decimal
-o	Words in octal
-x	Words in hexadecimal

The default base is octal on the PDP-11 and hexadecimal on the i80286, Z-8001, and M68000 families of microprocessors.

Dumping can start at position *offset* into the file. The specified *offset* is octal unless the '.' suffix is present to signify decimal. *offset* is in bytes unless the **b** suffix is present to signify 512-byte blocks.

See Also

ASCII, commands, conv, db, scat, strings

open() — COHERENT System Call (libc)

Open a file
int open(*file*, *type*) **char** ******file*; **int** *type*;

open opens a *file* to receive data, or to have its data read. When it opens *file*, **open** returns a file descriptor, which is a small, positive integer that identifies the open *file* for subsequent calls to **read**, **write**, **close**, **dup**, or **dup2**. *type* determines how the file is opened, as follows:

0	Read only
1	Write
2	Read and write

After *file* is opened, reading or writing begins at byte 0.

Example

This example copies the file named in **argv[1]** to the one named in **argv[2]** by using COHERENT system calls. It demonstrates the system calls **open**, **close**, **read**, **write**, and **creat**.

```
#include <stdio.h>
#define BUFSIZE (20*512)
char buf[BUFSIZE];

void fatal(s)
char *s;
{
	fprintf(stderr, "copy: %s\n", s);
	exit(1);
}

main(argc, argv)
int argc; char *argv[];
{
	register int ifd, ofd;
	register unsigned int n;

	if (argc != 3)
		fatal("Usage: copy source destination");
	if ((ifd = open(argv[1], 0)) == -1)
		fatal("cannot open input file");
	if ((ofd = creat(argv[2], 0666)) == -1)
		fatal("cannot open output file");

	while ((n = read(ifd, buf, BUFSIZE)) != 0) {
		if (n == -1)
			fatal("read error");
		if (write(ofd, buf, n) != n)
			fatal("write error");
	}

	if (close(ifd) == -1 || close(ofd) == -1)
		fatal("cannot close");
	exit(0);
}
```

See Also

COHERENT system calls, fopen()

Diagnostics

open returns -1 if the file is nonexistent, if the caller lacks permission, or if a system resource is exhausted.

Notes

open is a low-level call that passes data directly to COHERENT. It should not be mixed with high-level calls, such as **fread**, **fwrite**, or **fopen**.

operator — Definition

An **operator** is a function that is built into the C language. It usually relates one operand to another. For example, the statement

```
1+2
```

relates the operands **1** and **2** through the operation of addition; on the other hand, the statement

```
A>B
```

relates the operands **A** and **B** logically, by asserting that the former is greater than the latter; whereas

```
A=B
```

relates the operands **A** and **B** by assigning the value of the latter to the former. The following is a table of the C operators:

*	Multiplication
/	Division
%	Remainder
+	Addition
-	Subtraction
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
&&	Logical AND
!=	Inequality
!	Logical negation
| |	logical OR
&	Bitwise AND
^	Bitwise exclusive OR
~	Bitwise complement
|	Bitwise inclusive OR
<<	Bitwise shift left
>>	Bitwise shift right
=	Assign
+=	Increment and assign
-=	Decrement and assign
*=	Multiply and assign
/=	Divide and assign
%=	Modulus and assign
++	Increment
--	Decrement
==	Equivalence
&=	Bitwise AND and assign
^=	Bitwise exclusive OR and assign
|=	Bitwise inclusive OR and assign
<<=	Bitwise shift left and assign
>>=	Bitwise shift right and assign
*	Indirection
&	Render an address
()	Function indicator
[]	Array indicator
->	Structure pointer
.	Structure member
? :	Conditional expression

sizeof size of an object

Precedence

Precedence refers to the order in which C executes operators. The C languages assigns a level of precedence to each operator. Operators are executed in the order of their precedence level, from highest to lowest.

The following table summarizes the precedence of C operators. The are listed in *descending* order of precedence: those listed higher in the table are executed before those lower in the table. Operators listed on the same line have the same level of precedence, and the implementation determines the order in which they are executed. If you use two or more such operators in the same expression, you would be wise to use parentheses to indicate exactly the order in which you want the operators executed.

Operator	*Associativity*
`()  []  ->  .`	Left to right
`!  -  ++  --  -  (type)  *  &  sizeof`	Right to left
`*  /  %`	Left to right
`+  -`	Left to right
`<<  >>`	Left to right
`<  <=  >  >=`	Left to right
`==  !=`	Left to right
`&`	Left to right
`^`	Left to right
`|`	Left to right
`&&`	Left to right
`||`	Left to right
`?:`	Right to left
`=  +=  -=  *=  /=  %=`	Right to left
`,`	Left to right

You can always determine precedence in an expression by enclosing sub-expressions within parentheses: the expression enclosed within the innermost parentheses is always executed first.

See Also

definitions, sizeof

P

PAGER — Environmental Variable

Specify Output Filter
PAGER="*command options*"

The environmental variable **PAGER** directs programs such as **msgs**, **mail** and others to "pipe" their output into the *command* specified as the value of **PAGER**. For example, the following sets up **/bin/scat** as the desired output filter and passes a command line option to it to specify that the output screen has 20 lines.

```
export PAGER="exec /bin/scat -l20"
```

See Also

scat, environmental variables, mail, msgs

param.h — Header File

Define machine-specific parameters
#include <sys/param.h>

param.h defines machine-specific parameters. These parameters set limits on the operation of the COHERENT system; e.g., the number of files that can be open at any one time.

See Also

header files

passwd — File Format

Password file format

The file **/etc/passwd** holds information about each user who has permission to use the COHERENT system. This information is read by the commands **login** and **passwd** whenever a user attempts to log in, to ensure that that user is really himself and not an impostor.

/etc/passwd holds one record for each user; each record, in turn, consists of seven colon-separated fields, as follows:

```
name:password:user_id:group_id:comments:home_dir:shell
```

name is the user's login name. *password* is his encrypted password. *user_id* is a unique number that is also used to identify the user. *group_id* identifies the group to which the user belongs, if any. *comments* holds miscellaneous data, such as names, telephone numbers, or office numbers. *home_dir* gives the user's home directory. Finally, *shell* gives the program that is first executed when the user logs on; in most instances, this is an interactive shell (default, **/bin/sh**).

/etc/passwd includes a special entry for **remacc**. This entry controls access to the system by remote devices (for example, by a modem). If an entry in file **/etc/ttys** indicates that a serial line is remote (as set by placing an 'r' as the second character in its entry), COHERENT prompts

```
Remote access password:
```

when a user attempts to log in on that line.

To set the password for **remacc**, enter the following command while running as the superuser.

```
passwd remacc
```

See Also

file formats, passwd (command)

Notes

/etc/passwd can be read by anyone: if access to it were refused to a user, he could not log on. Thus, the passwords encrypted within it can be read and copied by anyone, and so may be vulnerable to brute-force decryption. For this reason, close attention should be paid to passwords: they should not be common words or names, preferably mix cases or use unique spellings, and be at least six characters long.

passwd — Command

Set/change login password
passwd [*user*]

passwd sets or changes the password for the specified *user*. If *user* is not specified, **passwd** changes the password of the caller.

passwd requests that the old password (if any) be typed, to ensure the caller is who he claims to be. Next it requests a new password, and then requests it again in case of typing errors. **passwd** will ask for a longer password if the one given is too short or not unusual enough.

Files

/etc/passwd — Encrypted passwords

See Also

commands, crypt(), login

Notes

One good way to construct a password is to concatenate two common words plus a punctuation mark. For example, "dog@collar" or "hamlet&horatio" are passwords that are both easy to remember and difficult to guess.

paste — Command

Merge lines of files
paste [-s] [-d *list*] *file* ...

paste merges corresponding lines from multiple input files. By default, **paste** uses the **<tab>** character to delineate texts from different files. **paste** writes the the merged text to standard output; thus, **paste** can be used at the head of a shell pipeline.

If **paste** reads EOF from any of the input files while other files still contain data, it substitutes blank lines as input from the file that has ended.

Options

paste recognizes the following command-line options:

-d *list*

Use the characters in *list* to separate the output fields. The characters in *list* are taken in sequence and used circularly, i.e., taken in order until the end of *list* is reached, then returning to the first character in *list*. Normally, **paste** uses the **<tab>** character to delineate the output fields. The following character sequences have special meaning when encountered in *list*:

\\ Output a single backslash character

\t Output a **<tab>** character
\n Output a **<newline>** character
\0 Output a null string (i.e., no separator between output fields)

-s Output successive lines from each input *file* across the page, with each input line separated from the next by a **<tab>** character. After all input lines from a given file have been concatenated, terminate the output line with a **<newline>** character and repeat the process on the next input file.

Example

The following two files will be used for subsequent examples. **File1** contains:

```
File1_Line1
File1_Line2
File1_Line3
File1_Line4
```

File2 contains:

```
File2_Line1
File2_Line2
File2_Line3
File2_Line4
```

The command

```
paste File1 File2
```

generates the following output:

```
File1_Line1      File2_Line1
File1_Line2      File2_Line2
File1_Line3      File2_Line3
File1_Line4      File2_Line4
```

Adding the option **-s** yields the output:

```
File1_Line1      File1_Line2      File1_Line3      File1_Line4
File2_Line1      File2_Line2      File2_Line3      File2_Line4
```

See Also

awk, commands, cut, sed

Notes

paste is copyright © 1989 by The Regents of the University of California. All rights reserved.

paste is distributed as a service to COHERENT customers, as is. It is not supported by Mark Williams Company. *Caveat utilitor.*

patch — Command

Modify portions of an executable
/conf/patch [-k] *image symbol=value ...*

The command **patch** alters the value of datum *symbol* to *value* in executable *image*. In general, you should use **patch** to alter configuration data (constants) in programs, in device drivers, and in the COHERENT kernel. For **patch** to work with a symbolic constant, *image* must have a symbol table that includes information about *symbol*. Therefore, executables

that have been processed by the command **strip** cannot be **patch**ed.

Option **-k** patches the kernel memory of the running COHERENT system via device **/dev/kmem**. Only the superuser **root** can access kernel memory from the **patch** command.

Both *symbol* and *value* may consist of numeric constants or a symbol from the symbol table of *image*. Note that trailing underscore ('_') characters are significant when specifying *symbol*. *symbol* and *value* expressions may include an optional numeric offset. In addition, the *value* field may optionally be composed of the construct **makedev(***major*,*minor***)**, where *major* and *minor* are the "major" and "minor" device numbers, respectively, resulting in a **dev_t**-sized device type.

Numeric constants default to decimal, but may be specifyed with a leading **0** prefix to specify an octal number or a **0x** prefix to specify a hexadecimal number.

The size of the altered *symbol* field is, by default, **sizeof(int)**. **patch** recognizes the following explicit size overrides:

:c The size of the altered field is **sizeof(char)**.

:i The size of the altered field is **sizeof(int)**.

:l The size of the altered field is **sizeof(long)**.

:s The size of the altered field is **sizeof(short)**.

Examples

The following example patches the value of kernel variable **KBBOOT_** in the **nkb** keyboard device driver to disallow rebooting of the system via the traditional **<ctrl><alt><del>** key sequence. See Lexicon article **nkb** for details. Note that this command changes **/coherent** on the boot device, not the copy of **/coherent** that is now running in memory.

```
/conf/patch /coherent KBBOOT_=0
```

The second example patches the value of character variable **myvar** in user-supplied program **myprog** to hexadecimal value 12. Note that the COHERENT C compiler appends a trailing underscore character ('_') to all C variable names.

```
/conf/patch myprog myvar_=0x12:c
```

The final example modifies the default "root" and "pipe" devices for COHERENT kernel **/testcoh** to be AT/IDE hard-disk partition **/dev/at0b**.

```
/conf/patch /testcoh 'rootdev_=makedev(11,1)' 'pipedev_=makedev(11,1)'
```

Note that in this last example, the arguments to **patch** must be quoted to avoid interpretation by the shell.

See Also

commands, device drivers

Notes

No spaces can appear around the equal sign in the **symbol=***constant* construct.

Using **patch** to modify the kernal data area of a running system is extremely dangerous. It should only be done by experienced writers of device drivers. *Caveat utilitor!*

/conf/patch /drv/at STIRQ=3

PATH — Environmental Variable

Directories that hold executable files

PATH names a default set of directories that are searched by COHERENT when it seeks an executable file. You can set **PATH** with the command **PATH**. For example, typing

```
PATH=/bin:/usr/bin
```

tells COHERENT to search for executable files first in **/bin**, and then in **/usr/bin**. Note the use of the colon ':' to separate directory names.

See Also

environmental variables, path.h

path() — General Function

Path name for a file

#include <path.h>
#include <stdio.h>
char *path(*path, filename, mode***);**
char ****path, *filename*;**
int *mode***;**

The function **path** builds a path name for a file.

path points to the list of directories to be searched for the file. You can use the function **getenv** to obtain the current definition of the environmental variable **PATH**, or use the default setting of **PATH** found in the header file **path.h**, or, you can define *path* by hand.

filename is the name of the file for which **path** is to search. *mode* is the mode in which you wish to access the file, as follows:

1 Execute the file
2 Write to the file
4 Read the file

path calls the function **access** to check the access status of *filename*. If **path** finds the file you requested and the file is available in the mode that you requested, it returns a pointer to a static area in which it has built the appropriate path name. It returns NULL if either *path* or *filename* are NULL, if the search failed, or if the requested file is not available in the correct mode.

Example

This example accepts a file name and a search mode. It then tries to find the file in one of the directories named in the **PATH** environmental variable.

```
#include <path.h>
#include <stdio.h>
#include <stdlib.h>
```

```
void
fatal(message)
char *message;
{
	fprintf(stderr, "%s\n", message);
	exit(1);
}

main(argc, argv)
int argc; char *argv[];
{
	char *env, *pathname;
	int mode;

	if (argc != 3)
		fatal("Usage: findpath filename mode");

	if(((mode=atoi(argv[2]))>4) || (mode==3) || (mode<1))
		fatal("modes: 1=execute, 2=write, 3=read");

	env = getenv("PATH");
	if ((pathname = path(env, argv[1], mode)) != NULL) {
		printf("PATH = %s\n", env);
		printf("pathname = %s\n", pathname);
		return;
	} else
		fatal("search failed");
}
```

See Also

access(), access.h, general functions, PATH, path.h

path.h — Header File

Define/declare constants and functions used with path
#include <path.h>

path.h declares constants used to handle the **path** environmental variable. These include, among others, the default path, the path separator, and the list separator. **path.h** also declares the function **path**.

See Also

header files, path(), PATH

paths — Technical Information

Routing data base for mail
/usr/lib/mail/paths

File **/usr/lib/mail/paths** holds the data base used by the COHERENT mail system to route mail. Each line gives routing information to a host, and has the following format:

```
host	route [cost]
```

host names a UUCP host. Because **smail** uses a binary-search algorithm when searching the data base for a given host name, the lines in **paths** must be sorted into ascending order. (See the Lexicon entry for **bsearch** for details on binary searches.) **smail** ignores case when

it searches **paths**, so you should convert each host name to lower case before you sort **paths**.

The *route* field details the route by which mail can travel from your system to *host*. Note that it includes a **printf**-style format string.

The optional field *cost* is used by the COHERENT mail system to decide whether to queue outbound UUCP mail, or to invoke **uucico** to deliver the mail immediately. If the *cost* is at or below **smail**'s "queueing threshold", then **smail** will attempt to deliver it immediately. This speeds mail delivery between hosts that enjoy an inexpensive UUCP link, such as a hard-wired line, and batches mail that must be sent over expensive media, such as long-distance telephone. If the *cost* field is absent, **smail** gives this host a cost value above that of its queueing threshold.

Example

The following gives a sample **paths** file for a COHERENT system that its owner has named **lepanto**.

```
friend     friend!%s  300
hubsys     hubsys!%s  95
lepanto    %s    0
lepanto.ampr.org      %s    0
smart-host hubsys!%s  95
widget     hubsys!widget!%s      95
```

As this file shows, **lepanto** is linked to systems **hubsys** and **friend**. The cost of 95 associated with **hubsys** is low, and is appropriate to a low-cost link, such as a hard-wired link; On the other hand, the cost of 300 associated with **friend** is high, which indicates that the connection with **friend** is high-cost, such as a long-distance telephone connection. If cost is 100 or greater, mail will be queued for latter delivery. A cost below 100 causes **smail** to attempt immediate delivery.

In this example, machine **lepanto** is registered in the **ampr.org** Amateur Packet Radio domain. Note that machine name **lepanto** appears in both conventional and domain forms in order to help resolve addressing.

In order to avoid having to maintain a huge data base, the owner of **lepanto** uses **hubsys** as a *smart host*. The **smart-host** designation in the **paths** data base signals **smail** to forward any mail that it doesn't know how to deliver onto site **hubsys**.

Finally, **lepanto** can use **hubsys** to pass mail on to **widget**. Thus, when **rmail** receives mail for system **widget**, it will transmit it to **hubsys** for forwarding. Note that **hubsys**'s administrator must have given **lepanto** permission to use it as a mail relay, or this would not work.

See Also

mail, technical information

pattern — Definition

A **pattern** is any combination of ASCII characters and wildcard characters that can be interpreted by a command.

The function **pnmatch** compares two patterns and signals if they match.

See Also

definitions, egrep, pnmatch(), wildcards

pause() — COHERENT System Call (libc)

Wait for signal
int pause()

pause suspends execution until the process receives a signal. The awaited signal could come from **kill**, **alarm**, or the controlling terminal.

See Also

alarm, COHERENT system calls, kill, signal(), sleep()

pax — Command

Backup: find . -print | pax -wv -f /dev/rfnva0
Restore: pax -rvoup -f /dev/rfnva0

Portable archive interchange

pax is an archiving utility that reads and writes **tar** and **cpio** formats, both the traditional ones and the extended formats specified in IEEE document 1003.1. It handles multi-volume archives and automatically determines the format of an archive while reading it.

pax supports three user interfaces: **tar**, **cpio**, and **pax**. The **pax** interface was designed by IEEE 1003.2 as a compromise in the chronic controversy over which of **tar** or **cpio** interfaces is superior.

See Also

commands, cpio, tar, ustar

Notes

To avoid confusion with the traditional COHERENT **tar** command, the **tar** command distributed with **pax** is renamed **ustar**.

See the compressed **tar** archive **/usr/src/alien/pax.tar.Z** for full documentation on **pax**, **cpio**, and **ustar**.

pax was developed by Mark H. Colburn and sponsored by The USENIX Association. It is provided in binary form per the licensing terms set forth by the author. See file **/usr/src/alien/pax.tar.Z** for licensing terms. Copyright © 1989 by Mark H. Colburn. All rights reserved.

pclose() — STDIO Function (libc)

Close a pipe
#include <stdio.h>
int pclose(*fp*)
FILE **fp*;

pclose closes the pipe pointed to by *fp*, which must have been opened by the function **popen**.

pclose awaits the completion of the child process and performs other cleanup. It returns the value from a **WAIT** done on the child process. This value includes information in addition to the "simple" exit value of the child process.

Files

<stdio.h>

See Also

fclose(), fopen(), pipe(), popen(), sh, STDIO, system(), wait()

Diagnostics

pclose returns -1 if *fp* had not been created by a call to **popen**. Otherwise, **pclose** returns the exit status of the *command*, in the format described in the entry for **wait**: exit status in the high byte, signal information in the low byte.

Permissions — File Format

Format of UUCP permissions file
/usr/lib/uucp/Permissions

The file **Permissions** describes the remote sites that can communicate via UUCP with your COHERENT system, and lists the programs that each site can execute on your system. Before a remote site can communicate with your COHERENT system, that site must have an entry in **Permissions**.

When the command **uucico** attempts to execute a file transfer to or from a remote site, it checks to see that there is an entry for the site in **Permissions**. If your **Permissions** entries are not written correctly, you risk a breach of system security.

Each entry in **Permissions** takes one of two forms:

- **LOGNAME** entries detail the permissions granted to an individual user when he calls your system from a remote site.
- **MACHINE** entries detail the permissions for the remote sites that you call.

You can combine the two types of entries into one entry if the permissions are the same in both entries.

An entry in **Permissions** consists of pairs of entries of the form *OPTION=value*, each separated by one or more white-space characters. The *OPTION* side must be in upper-case characters, and the *value* side in lower-case characters. At the end of each line (except the last), you must include a backslash character ('\') to continue the current line onto the next one. Blank lines between entries are ignored.

For the **READ**, **NOREAD**, **WRITE**, and **NOWRITE** fields, described below, the value specified is a sequence of one or more directories on your computer, separated only by colons (i.e., no white space allowed).

An entry in **Permissions** can have up to ten fields:

1. **MACHINE**
   This field names the remote system that you wish to communicate with. It is limited to seven characters. (Future releases of COHERENT will increase this limit.)

2. **LOGNAME**
   This field specifies the login name that the remote system will use when it calls your system. Please note that if the remote site attempts to log into your system with a login name other than the one specified by this field, **uucico** will terminate the call for security reasons. There must be a valid entry in file **/etc/passwd** for the name specified in this field.

3. **READ**
   This entry names the directories on your system that the UUCP commands can access. You must give the full path name of the directory. The default is **/usr/spool/uucppublic**.

4. **NOREAD**
When a directory is entered in the **READ** field, all of its sub-directories become available for reading. If you wish to make any of its sub-directories unreadable by the remote site, name it here. You must give the full path name of the directory. The default is NULL.

5. **WRITE**
Here, name the directories on your system into which the command **uucico** can deposit files. You must give the full path name of the directory. The default is **/usr/spool/uucppublic**.

6. **NOWRITE**
When a directory is entered in the **WRITE** field, the remote system can write into all of its sub-directories. If you wish to make any of its sub-directories unwriteable by the remote site, enter it here. You must give the full path name of the directory. The default is NULL.

7. **COMMANDS**
Here, name the commands that the remote system can execute on your computer. The two most basic commands to put in this entry are **rmail** and **uucp**. This lets the remote site send electronic mail to you and to use uucp to transfer files. You may add other commands, but the shorter the list, the greater your level of system security. The default is **rmail**.

8. **REQUEST**
This entry asks if the remote site can request to transfer files from your system. Respond **yes** if security is not an issue. If the value is **no**, only your system can request that files be transfered to the remote system. The default is **no**.

9. **SENDFILES**
This entry asks if your system can initiate file transfers to the remote site. Your response can be **yes**, **no**, or **call**. The default is **call**, which allows files to be sent only when your system calls the remote site. A value of **yes** allows your system to transfer files to the remote system regardless of which system originated the conversation. A value of **no** prohibits any file transfers from your system to the remote system.

10. **MYNAME**
This field contains the site name that you have been assigned by the system administrator of the remote site. It must contain no more than seven characters. If **MYNAME** is defined, its value is used as your site name rather than the value in **/etc/uucpname**. This is useful in situations where your site name is already used by an existing account on the remote site you wish to call, or when the remote site does not support "anonymous" UUCP access.

When writing your Permissions file, keep these considerations in mind:

- White space is not allowed before or after the '=' sign.
- Each line corresponds to one entry. You may continue to the next line by ending the line with a backslash charcter ('\').
- If a field has more than one value, use a colon to separate them.

Example

The following example gives an entry in **Permissions** to set up a connection with the Mark Williams Company's UUCP BBS:

```
MACHINE=mwcbbs MYNAME=bbsuser \
REQUEST=yes SENDFILES=yes \
COMMANDS=rmail:uucp \
READ=/usr/spool/uucppublic:/tmp \
WRITE=/usr/spool/uucppublic:/tmp
```

See Also

file formats, UUCP

perror() — General Function (libc)

System call error messages
#include <errno.h>
perror(*string*)
char *****_string_**; extern int** _sys_nerr_**; extern char** *****_sys_errlist_**[];**

perror prints an error message on the standard error device. The message consists of the argument *string*, followed by a brief description of the last system call that failed. The external variable **errno** contains the last error number. Normally, *string* is the perror of the command that failed or a file perror.

The external array **sys_errlist** gives the list of messages used by **perror**. The external **sys_nerr** gives the number of messages in the list.

See Also

errno, errno.h, general functions

phone — Command

Print numbers and addresses from phone directory
phone *person* ...

The command **phone** searches a number of telephone directory files for each *person* argument that is given. Any lines that matches any of the *person* arguments is printed. Typically, such lines contain the telephone number, name, and address of a person or organization. Lower-case letters in *person* can be matched by both the same letter and the corresponding upper-case letter in the phone directory.

The user may supply his own phone directory by setting the (exported) shell variable **PHONEBOOK**, to the name of that file. If given, this file is searched first. Then, the system-wide phone book is always searched.

Files

$PHONEBOOK — User-supplied phonebook (searched first)
/usr/pub/phonebook — System-wide phone directory

See Also

commands

Diagnostics

phone exits with non-zero status if a call fails. A diagnostic message is written to **stderr** if no matching entries are found.

pipe() — COHERENT System Call

Open a pipe
int pipe(*fd*)
int *fd*[2];

A *pipe* is an interprocess communication mechanism. **pipe** creates a pipe, typically to construct pipelines in the shell **sh**.

pipe fills in *fd*[0] and *fd*[1] with *read* and *write* file descriptors, respectively. The file descriptors allow the transfer of data from one or more writers to one or more readers. Pipes are buffered to 5,120 bytes. If more than 5,120 bytes are written into the pipe, the **write** call will not return until the reader has removed sufficient data for the **write** to complete. If a **read** occurs on an empty pipe, its completion awaits the writing of data.

When all writing processes close their write file descriptors, the reader receives an end of file indication. A write on a pipe with no remaining readers generates a **SIGPIPE** signal to the caller.

pipe is generally called just before **fork**. Once the parent and child processes are created, the unused file descriptors should be closed in each process.

Example

The following example prints the word **Waiting** until a line of data is entered. It illustrates how to use **pipe**, **fstat**, and **fork**.

```
/*
 * For greatest effeciency compile with -n option.
 * This causes separate code and data segments, when the
 * fork() occurs there will be one code and 2 data segments.
 * Use of the -n option prevents db from setting breakpoints.
 */

#include <stdio.h>
#include <sys/stat.h>      /* for stat */
#include <sgtty.h>         /* for stty/gtty functions */

static int fd[2];          /* pipe array */

main()
{
    printf("This prints 'Waiting' every second until a 'q' is hit.\n");

    /*
     * Pipe may also be constructed by /etc/mknod
     * If it is desired to have tasks communicate where
     * they are not parent and child. In this case make
     * sure the constructed pipe has the correct owner and
     * permissions. Such pipe may be used exactly like this
     * but open()ed on each side.
     */
```

```
    if (-1 == pipe(fd)) {
        fprintf(stderr, "Cannot open pipe\n");
        exit(1);
    }

    if (fork())
        parentProcess();
    else
        childProcess();
    exit(0);
}

parentProcess()
{
    struct stat s;
    char buff;

    for (buff = ' '; 'q' != buff;) {
        fstat(fd[0], &s);     /* get status of pipe */
        if (s.st_size) {          /* char in the pipe */
            read(fd[0], &buff, sizeof(buff));
            printf("Got a '%c'\n", buff);
            continue;
        }
        /*
         * This can be any process, it can use system()
         * or exec()
         */
        printf("Waiting\n");
        sleep(1);
    }
}

childProcess()
{
    struct    sgttyb    os, ns;
    char buff;

    gtty(fileno(stdin), &os);  /* save old state */
    ns = os;                   /* get base of new state */
    ns.sg_flags |= RAW;        /* process each character as entered */
    ns.sg_flags &= ~(ECHO|CRMOD);   /* no echo for now... */
    stty(fileno(stdin), &ns);  /* set mode */

    do {
        buff = getchar();     /* wait for the keyboard */
        write(fd[1], &buff, sizeof(buff));
    } while ('q' != buff);

    stty(fileno(stdin), &os);  /* reset mode */
}
```

See Also

close(), COHERENT system calls, mknod(), read(), sh, signal(), write()

Diagnostics

pipe returns zero on successful calls, or -1 if it could not create the pipe.

If it is necessary to create a pipe between tasks that are not parent and child, use **/etc/mknod** to create a named pipe. These named pipes can be opened and used by different programs for communication. Remember to give them the correct owner and permissions.

pipe — Definition

A *pipe* directs the output stream of one program into the input stream of another program, thus coupling the programs together. With pipes, two or more programs (or *filters*) can be coupled together to perform complex transforms on streams of data. For example, in the following command

```
cat DATAFILE1 DATAFILE2 | sort | uniq -d
```

the filter **cat** opens two files and prints their contents. Its output is piped to the filter **sort**, which sorts it. The output of **sort** is piped, in turn, to the filter **uniq**, which (with the **-d** option) prints a single copy of each line that is duplicated within the file. Thus, with this simple set of commands and pipes, a user can quickly print a list of all lines that appear in both files.

See Also

definitions, filter, named pipe

pnmatch() — String Function (libc)

Match string pattern
int pnmatch(*string*, *pattern*, *flag*)
char **string*, **pattern*; **int** *flag*;

pnmatch matches *string* with *pattern*, which is a regular expression. The shell **sh** uses patterns for file name expansion and **case** statement expressions.

pnmatch returns one if *pattern* matches *string*, and zero if it does not. Each character in *pattern* must exactly match a character in *string*; however, the wildcards '*', '?', '[' and ']', and '[!' and ']' can be used in *pattern* to expand the range of matching.

flag must be either zero or one: zero means that *pattern* must match *string* exactly, whereas one means that *pattern* can match any part of *string*. In the latter case, the wildcards '' and '$' can also be used in *pattern*.

Example

For an example of this function, see the entry for **fgets**.

See Also

egrep, general functions, grep, sh

Notes

flag must be zero or one for **pnmatch** to yield predictable results.

pnmatch is a more powerful version of the ANSI functions **strstr** and **strcmp**.

pointer — C Language

A *pointer* is an object whose value is the address of another object. The name "pointer" derives from the fact that its contents "point to" another object. A pointer may point to any type, complete or incomplete, including another pointer. It may also point to a function, or to nowhere.

The term *pointer type* refers to the object of a pointer. The object to which a pointer points is called the *referenced type*. For example, an **int** * ("pointer to **int**") is a pointer type; the referenced type is **int**. Constructing a pointer type from a referenced type is called *pointer type derivation*.

The Null Pointer

A pointer that points to nowhere is a *null pointer*. The macro **NULL**, which is defined in the header **stdio.h**, defines the null pointer. The null pointer is an integer constant with the value zero. It compares unequal to a pointer to any object or function.

Declaring a Pointer

To declare a pointer, use the indirection operator '*'. For example, the declaration

```
int *pointer;
```

declares that the variable **pointer** holds the address of an **int**-length object. Likewise, the declaration

```
int **pointer;
```

declares that **pointer** holds the address of a pointer whose contents, in turn, point to an **int**-length object.

Failure to declare a function that returns a pointer will result in that function being implicitly declared as an **int**. This will not cause an error on microprocessors in which an **int** and a pointer have the same size; however, transporting this code to a microprocessor in which an **int** consists of 16 bits and a pointer consists of 32 bits will result in the pointers being truncated to 16 bits and the program probably failing.

C allows pointers and integers to be compared or converted to each other without restriction. The COHERENT C compiler flags such conversions with the strict message

```
integer pointer pun
```

and comparisons with the strict message

```
integer pointer comparison
```

These problems should be corrected if you want your code to be portable to other computing environments.

See for more information.

Wild Pointers

Pointers are omnipresent in C. C also allows you to use a pointer to read or write the object to which the pointer points; this is called *pointer dereferencing*. Because a pointer can point to any place within memory, it is possible to write C code that generates unpredictable results, corrupts itself, or even obliterates the operating system if running in unprotected mode. A pointer that aims where it ought not is called a *wild pointer*.

When a program declares a pointer, space is set aside in memory for it. However, this space has not yet been filled with the address of an object. To fill a pointer with the address of the object you wish to access is called *initializing* it. A wild pointer, as often as not, is one that

is not properly initialized.

Normally, to initialize a pointer means to fill it with a meaningful address. For example, the following initializes a pointer:

```
int number;
int *pointer;
   . . .
pointer = &number;
```

The address operator '&' specifies that you want the address of an object rather than its contents. Thus, **pointer** is filled with the address of **number**, and it can now be used to access the contents of **number**.

The initialization of a string is somewhat different than the initialization of a pointer to an integer object. For example,

```
char *string = "This is a string."
```

declares that **string** is a pointer to a **char**. It then stores the string literal **This is a string** in memory and fills **string** with the address of its first character. **string** can then be passed to functions to access the string, or you can step through the string by incrementing **string** until its contents point to the null character at the end of the string.

Another way to initialize a pointer is to fill it with a value returned by a function that returns a pointer. For example, the code

```
extern char *malloc(size_t variable);
char *example;
   . . .
example = malloc(50);
```

uses the function **malloc** to allocate 50 bytes of dynamic memory and then initializes **example** to the address that **malloc** returns.

Reading What a Pointer Points To

The indirection operator '*' can be used to read the object to which a pointer points. For example,

```
int number;
int *pointer;
   . . .
pointer = &number;
   . . .
printf("%d\n", *pointer);
```

uses **pointer** to access the contents of **number**.

When a pointer points to a structure, the elements within the structure can be read by using the structure offset operator '->'. See the entry for **operators** for more information.

Pointers to Functions

A pointer can also contain the address of a function. For example,

```
char *(*example)();
```

declares **example** to be a pointer to a function that returns a pointer to a **char**.

This declaration is quite different from:

```
char **different();
```

The latter declares that **different** is a function that returns a pointer to a pointer to a **char**.

The following demonstrates how to call a function via a pointer:

```
(*example)(arg1, arg2);
```

Here, the '*' takes the contents of the pointer, which in this case is the address of the function, and uses that address to pass to a function its list of arguments.

A pointer to a function can be passed as an argument to another function. The functions **bsearch** and **qsort** both take function pointers as arguments. A program may also use of arrays of pointers to functions.

Pointer Conversion

One type of pointer may be converted, or *cast*, to another. For example, a pointer to a **char** may be cast to a pointer to an **int**, and vice versa.

Pointers to different data types are compatible in expressions, but only if they are cast appropriately. Using them without casting produces a *pointer-type mismatch*.

Pointer Arithmetic

Arithmetic may be performed on all pointers to scalar types, i.e., pointers to **chars** or **int**. Pointer arithmetic is quite limited and consists of the following:

1. One pointer may be subtracted from another.
2. An **int** or a **long**, either variable or constant, may be added to a pointer or subtracted from it.
3. The operators ++ or -- may be used to increment or decrement a pointer.

No other pointer arithmetic is permitted. No arithmetic can be performed on pointers to non-scalar objects, e.g., pointers to functions.

When an **int** or **long** is added to a pointer, it is first multiplied by the length of what the pointer is declared as pointing to. Thus, if a pointer to an **int** is incremented by two, it points down two more **ints**, not two more characters. The following program demonstrates this feature:

```
char *pc = "Welcome";
int array[5] = { 1, 2, 3, 4, 5 };
int  *pi = array;

main()
{
      pc += 2;      /* pc points to  'l' */
      pi += 2;      /* pi points to 3 */
}
```

i8086 Pointers

Intel designed the i8086 to use a segmented architecture. This means that the i8086 divides memory into 64-kilobyte segments. To program the i8086 requires that you use a specific *memory model*, which describes how the segments of memory are to be organized.

See Also

C language, data formats, operators, portability

poll.h — Header File

Define structures/constants used with polling devices
#include <sys/poll.h>

poll.h defines structures and constants used by routines that poll devices.

See Also

header files

popd — Command

Pop an item from the directory stack
popd [*item* ...]

The COHERENT shell **sh** maintains an internal "directory stack", which is a stack of names of directories. You can manipulate this stack should you, for any reason, wish to traverse a number of directories quickly and efficiently.

The command **popd** pops an item from the directory stack. If called without an argument, it pops the last item. Otherwise, it pops the given stack *items* in the order requested, where each *item* is a positive integer and zero is the top of the stack.

See Also

commands, dirs, pushd, sh

popen() — STDIO Function (libc)

Open a pipe
#include <stdio.h>
FILE *popen(*command, how***)**
**char ****command*, **how*;

popen opens a pipe. It resembles the function **fopen**, except that the opened object is a command line to the shell **sh** rather than a file.

The caller can read the standard output of *command* when *how* is **r**, or write to the standard input of *command* when *how* is **w**. **popen** returns a pointer to a **FILE** structure that may be read or written.

Files

<stdio.h>

See Also

fclose(), fopen(), pclose(), pipe(), sh, STDIO, system(), wait()

Diagnostics

popen returns NULL if the link to *command* could not be established.

port — Definition

A **port** passes data to and receives data from a remote device.

See Also

definitions, FILE, stream

portability — Technical Information

Portability means that code can be recompiled and run under different computing environments without modification. Although true portability is an ideal that is difficult to realize, you can take a number of practical steps to ensure that your code is portable:

- Do not assume that an integer and a pointer have the same size. Remember that undeclared functions are assumed to return an **int**. If a function returns a pointer, declare it so.
- Do not write routines that depend on a particular order of code evaluation, particular byte ordering, or particular length of data types.
- Do not write routines that play tricks with a machine's "magic characters"; for example, writing a routine that depends on a file's ending with **<ctrl-Z>** instead of **EOF** ensures that that code can run only under operating systems that recognize this magic character.
- Always use manifest constants, such as **EOF**, and make full use of **#define** statements.
- Use header files to hold all machine-dependent declarations and definitions.
- Declare everything explicitly. In particular, be sure to declare functions as **void** if they do not return a value; this avoids unforeseen problems with undefined return values.
- Do not assume that integers and pointers have the same size or even the same kind of structure. Do not assume that pointers are all the same or can point anywhere. On the i8086, in SMALL model a pointer to a function addresses relative to the code segment, whereas a pointer to data addresses relative to the data segment. On some machines, character pointers are of a different size or structure than word pointers.
- The constant NULL is defined as being different from any valid pointer. Use it and nothing else for that purpose.
- Keep test scripts, preferably at the function level. That is, follow each function with an

  ```
  #ifdef TEST
  ```

  section that will exercise that function. Running these can rapidly isolate portability problems.
- Place plenty of

  ```
  #assert
  ```

  statements in your programs. These can often pick up portability problems.

See Also

header file, pointer, technical information, void

pow() — Multiple-Precision Mathematics

Raise multiple-precision integer to power
#include <mprec.h>
void pow(*a*, *b*, *m*, *c*)
mint *****a*, *****b*, *****m*, *****c*;

The COHERENT system includes a suite of routines that allow you to perform multiple-precision mathematics. The function **pow** sets the multiple-precision integer (or **mint**) pointed to by *c* to the value pointed to by *a* raised to the power of the value pointed to by *b*,

reduced modulo of the value pointed to by *m*.

See Also

multiple-precision mathematics

pow() — Mathematics Function (libm)

Compute a power of a number
#include <math.h>
double pow(*z*, *x*) **double** *z*, *x*;

pow returns *z* raised to the power of *x*, or z^x.

Example

For an example of this function, see the entry for **exp**.

See Also

mathematics library

Diagnostics

pow indicates overflow by an **errno** of **ERANGE** and a huge returned value.

pr — Command

Paginate and print files
pr [*options*] [*file* ...]

pr paginates each named *file* and sends it to the standard output. The file name '-' means standard input. If no *file* is named, **pr** reads the standard input.

Each page has a header that gives the date, file name, and page and line numbers. **pr** may be used with the following options.

+ *skip* Skip the first *skip* pages of each input file.

-*N* Print the text in *N* columns. This is used to print out material that was typed in one or more columns.

-h *header*
Use *header* in place of the text name in the title. If *header* is more than one word long, it must be enclosed in quotation marks.

-l*N* Set the page length to *N* lines (default, 66).

-m Print the texts simultaneously, in separate columns. Each text will be assigned an equal amount of width on the page, and any lines longer than that width will be truncated. This is used to print several similar texts or listings simultaneously.

-n Number each line as it is printed.

-s*c* Separate each column by the character *c*. You can separate columns with a letter of the alphabet, a period, or an asterisk. Normally, each column is left justified in a fixed-width field.

-t Suppress the printing of the header on each page, and the header and footer space.

-w*N* Set the page width to *N* columns (default, 80). Text lines are truncated to fit the column width. The maximum width is 254 columns.

See Also

cat, commands, nroff, prps

Diagnostics

Messages are written on the standard error.

prep — Command

Produce a word list

prep [**-dfp**] [**-i** *ifile*] [**-o** *ofile*] [*file* ...]

The command **prep** prepares a word list that is useful for statistical processing from the textual data found in each input *file*. If no *file* is given, **prep** reads the standard input for text.

For the purposes of **prep**, a word consists of a string of alphabetic letters and apostrophes. Words are written, one per line, to the standard output. Hyphenated words are treated as two words. However, any word hyphenated between two lines is rejoined as one word.

prep recognizes the following options:

-d Print a sequence number (of words in the input text) before each output word.

-f Fold upper-case letters into lower case. This is sometimes useful for producing unique lists of words.

-i *ifile* Ignore words found in *ifile*. *ifile* has words one per line that are matched again each input word independent of case.

-o *ofile* Print only words found in *ofile*. Only one of **-i** or **-o** may be specified.

-p In addition to printing words, also print each punctuation character (printable, non-numeric characters that separate words), one per line. These lines are not counted for **-d**.

See Also

commands, deroff, ksh, sh, sort, spell, typo, wc

Notes

What constitutes a *word* is different in **deroff**, **prep**, and **wc**.

print — Command

Echo text onto the standard output

print [**-enrun**] [*argument* ...]

The command **print** is built into the Korn shell **ksh**. It echoes each *argument* onto the standard output. Arguments are separated from each other by whitespace, and the list of arguments is terminated by a newline character.

print recognizes and substitutes for the following C-style escape sequences:

\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Tab
\v	Vertical tab
\0*nnn*	*nnn* is the octal value of the desired character

print recognizes the following options:

-e Re-enable expansion of C escape sequences.

-n Suppress printing of a newline at the end of the list of arguments.

-r Suppress expansion of C escape sequences.

-u*n* Redirect output from the standard output to shell file descriptor *n*.

See Also

commands, echo, ksh

printer — Technical Information

The printer is the device that transfers human-readable data to paper. It can be plugged into either a parallel or a serial port, depending upon how your printer is designed. The former is faster, whereas the latter permits the printer to be positioned farther away from the computer. The following descriptions assume that you have your printer plugged into a parallel port.

COHERENT permits you to have up to three parallel ports on your computer. Devices **/dev/lpt1**, **/dev/lpt2**, and **/dev/lpt3** control, respectively, parallel ports 1, 2, and 3 in cooked mode. The device **/dev/lp** is normally linked to the above device that you wish to use by default as your line printer. See the Lexicon article **lp** for more details on these devices.

COHERENT can print text on all "dumb" printers that have no special text-formatting features. It also supports text formatting on three varieties of printers: Epson-compatible dot-matrix printers; laser printers compatible with the Hewlett-Packard LaserJet family of printers that implement the Hewlett-Packard Page Control Language (PCL); and all printers that have implemented the PostScript language.

Dumb Printers

To print on a "dumb" printer plugged into the parallel port, use the command **lpr**. This command performs some formatting on a file, and invokes the line-printer daemon **lpd** to spool the file for printing. Using the line-printer daemon is necessary in a multi-user environment to ensure that print requests from different users do not arrive at the printer at the same time, causing the printer to output a jumbled mess (if it prints anything at all).

For example, if **FOO** is a text file, the command

```
lpr FOO
```

prints it on your dumb printer. You should use the **lpr** command to print "simple" text (such as program listings) on any variety of dot-matrix printer. To print listings or other simple text on a laser printer, see below.

The output of the text-formatting command **nroff** can also be printed, with some success, on dumb printers. To represent an italicized character, it prints the character, followed by a backspace, followed by an underscore character; to represent a bold-face character, it output the character, followed by a backspace, followed by the character again (in the hope, perhaps naive, that presenting the same text twice will make it appear bolder).

Epson-Compatible Printers

The command **epson** massages text into a form that uses some of the text-formatting features of the Epson MX-80 printer and clones thereof. It is especially to be used with text that has been formatted with **nroff**, as described above; there, it turns the "character/backspace/character" sequence into the Epson escape sequences for

emphasized text and italics. It then directs its output to the line-printer device **/dev/lp**, which it assumes has an Epson-style printer plugged into it.

The following example uses **nroff** to format file **FOO** and prints the output on an Epson-style printer:

```
nroff -ms FOO | epson
```

LaserJet-style Printers

COHERENT includes a large suite of commands to support the Hewlett-Packard LaserJet family of printers, as well as clones that run Hewlett-Packard's PCL.

To begin, these commands use the HP devices **/dev/hp** and **/dev/rhp**. When you installed COHERENT on your system, you may have created these devices; if you did not, however, you should create them by simply using the command **ln** to link **/dev/lp** to **/dev/hp** and to link **/dev/rlp** to **/dev/rhp**, as follows:

```
ln /dev/lp  /dev/hp
ln /dev/rlp /dev/rhp
```

You must log in as the superuser **root** to execute these commands.

The daemon **hpd** spools files to be printed on your laser printer. It works like the line-printer daemon **lpd**, as described above.

The command **hp** prepares files to be printed on a laser printer. You should use it to prepare "simple" text, such as program listings, for printing on your laser printer. Like the command **epson**, **hp** also massages the output of **nroff** into PCL-style escape sequences; unlike **epson**, however, it does not automatically spool the file for printing.

The command **hpr** spools files to be printed on a laser printer. It works like the command **lpr**, except that it includes a number of special features; for example, you can use it to download LaserJet "soft fonts" into your printer.

The following command uses **nroff** to format file **FOO**, then prints on a Hewlett-Packard style laser printer:

```
nroff -ms FOO | hp | hpr -B
```

Note that the **-B** option to **hpr** suppresses the printing of a banner page.

The text-formatting command **troff** can create proportionally spaced text to be printed on either a PCL or PostScript printer. In PCL mode, **troff** can make full use of all "soft fonts" that you have loaded onto your printer. For example, this manual was printed by COHERENT **troff** in PCL mode driving a Hewlett-Packard LaserJet III with soft fonts. See the Lexicon for details on how to use **troff** with laser printers.

PostScript Printers

COHERENT includes two commands that can drive PostScript style printers, such as the Apple LaserWriter.

The command **prps** is a PostScript version of the COHERENT command **pr**. It paginates text, and supplies each page with a simple header. See its Lexicon entry for details.

As noted above, **troff**, the COHERENT text formatter, can create proportionally space text for either PCL or PostScript printers. In PostScript mode, **troff** can handle all 35 fonts available with most PostScript cartridges; it supports full font scaling and features such as outlining and shadowing. It also permits you to embed "raw" PostScript within your file, to create effects not already available with the **troff** text-formatting language. For details on using **troff** with PostScript printers, see its entry in the Lexicon.

Printer Problems

The following paragraphs describes the problems most commonly encountered with printers, and suggests some solutions.

If you are trying to access your parallel interface printer via special files **/dev/lp** or **/dev/lpt1** and receive an error message of the form

```
cannot open device /dev/lp
```

this means that your printer is not attached to the device that COHERENT associates with **/dev/lpt1**. Your printer is at either attached to **/dev/lpt2** or to **/dev/lpt3**. To discover which one, log in as the superuser **root** and use **cd** to enter directory **/dev**. Make sure that your printer is plugged in, turned on, and on-line; then enter the command:

```
cat file > lpt2
```

file can be any readable file that you specify (e.g., **/etc/passwd**). If your printer does not print *file*, then repeat the command for device **/dev/lpt3**:

```
cat file > lpt3
```

The command that works indicates the device into which your printer is plugged.

The final step is to "link" the actual location of the printer to devices **/dev/lp** and **/dev/rlp**, so that the COHERENT utilities know how to print a file. Enter the appropriate commands:

```
ln -f lpt2  lp
ln -f rlpt2 rlp
```

if your printer is attached to **/dev/lpt2**, or

```
ln -f lpt3  lp
ln -f rlpt3 rlp
```

if your printer was attached to **/dev/lpt3**.

If you have an Hewlett-Packard LaserJet or compatible printer, perform the above "link" operation again but substitute **hp** for **lp** and **rhp** for **rlp**. This allows the command **hpr** to find your printer.

If you are using a serial printer, note that flow control via CTS (clear-to-send) is not supported in the **com1** through **com4** family of devices, but is available in devices **hs00r** through **hs07r**. See Lexicon articles **com**, **hs**, and **terminal** for details.

See Also

epson, hp, hpd, hpr, lp, lpd, lpr, prps, technical information, troff

printf() — STDIO (libc)

Print formatted text

int printf(*format* **[,***arg1*, ... *argN***])**
char ****format*; [data type]** *arg1*, ... *argN*;

printf prints formatted text. It uses the *format* string to specify an output format for each *arg*, which it then writes on the standard output.

printf reads characters from *format* one at a time; any character other than a percent sign '%' or a string that is introduced with a percent sign is copied directly to the output. A '%' tells **printf** that what follows specifies how the corresponding *arg* is to be formatted; the characters that follow '%' can set the output width and the type of conversion desired. The

following modifiers, in this order, may precede the conversion type:

1. A minus sign '-' left-justifies the output field, instead of the default right justify.
2. A string of digits gives the *width* of the output field. Normally, **printf** pads the fiel padded with spaces to the field width; it is padded on the left unless left justification is specified with a '-'. If the field width begins with '0', the field is padded with '0' characters instead of spaces; the '0' does not cause the field width to be taken as an octal number. If the width specification is an asterisk '*', the routine uses the next *arg* as an integer that gives the width of the field.
3. A period '.' followed by one or more digits gives the *precision*. For floating point (**e**, **f**, and **g**) conversions, the precision sets the number of digits printed after the decimal point. For string (**s**) conversions, the precision sets the maximum number of characters that can be used from the string. If the precision specification is given as an asterisk '*', the routine uses the next *arg* as an integer that gives the precision.
4. The letter '**l**' before any integer conversion (**d**, **o**, **x**, or **u**) indicates that the argument is a **long** rather than an **int**. Capitalizing the conversion type has the same effect; note, however, that capitalized conversion types are *not* compatible with all C compiler libraries, or with the ANSI standard. This feature will not be supported in future editions of COHERENT.

The following format conversions are recognized:

% Print a '%' character. No arguments are processed.

c Print the **int** argument as a character.

d Print the **int** argument as signed decimal numerals.

D Print the **long** argument as signed decimal numerals.

e Print the **float** or **double** argument in exponential form. The format is *d.ddddddesdd*, where there is always one digit before the decimal point and as many as the *precision* digits after it (default, six). The exponent sign *s* may be either '+' or '-'.

f Print the **float** or **double** argument as a string with an optional leading minus sign '-', at least one decimal digit, a decimal point ('.'), and optional decimal digits after the decimal point. The number of digits after the decimal point is the *precision* (default, six).

g Print the **float** or **double** argument as whichever of the formats **d**, **e**, or **f** loses no significant precision and takes the least space.

o Print the **int** argument in unsigned octal numerals.

O Print the **long** argument in unsigned octal numerals.

r The next argument points to an array of new arguments that may be used recursively. The first argument of the list is a **char** * that contains a new format string. When the list is exhausted, the routine continues from where it left off in the original format string.

s Print the string to which the **char** * argument points. Reaching either the end of the string, indicated by a null character, or the specified *precision*, will terminate output. If no *precision* is given, only the end of the string will terminate.

u Print the **int** argument in unsigned decimal numerals.

U Print the **long** argument in unsigned decimal numerals.

x Print the **int** argument in unsigned hexadecimal numerals.

X Print the **long** argument in unsigned hexadecimal numerals.

Example

The following example demonstrates many **printf** statements.

```
main()
{
     extern void demo_r();
     int precision = 1;
     int integer = 10;
     float decimal = 2.75;
     double bigdec = 27590.21;
     char letter = 'K';
     char buffer[20];

     strcpy (buffer, "This is a string.\n");

     printf("This is an int:   %d\n", integer);
     printf("This is a float:  %f\n", decimal);
     printf("Another float:    %3.*f\n", precision, decimal);
     printf("This is a double: %lf\n", bigdec);
     printf("This is a char:   %c\n", letter);
     printf("%s", buffer);
     printf("%s\n", "This is also a string.");

     demo_r("Print everything: %d %f %lf %c",
          integer, decimal, bigdec, letter);
     exit(0);
}

void demo_r(string)
char *string;
{
     printf("%r\n", (char **)&string);
}
```

See Also

fprintf(), putc(), puts(), scanf(), sprintf(), STDIO

Notes

Because C does not perform type checking, it is essential that each argument match its counterpart in the format string.

The use of upper-case format characters to specify long arguments is not standard, and will be phased out to conform with the ANSI standard. You should use the 'l' modifier to indicate a **long**.

At present, **printf** does not return a meaningful value.

proc.h — Header File

Define structures/constants used with processes
#include <sys/proc.h>

proc.h defines structures and constants used by routines that manipulate processes.

See Also

header files

process — Definition

A **process** is a program in the state of execution.

See Also

daemon, definitions, file

prof — Command

Print execution profile of a C program
prof [**-abcs**][*progfile* [*monfile*]]

prof interprets the profile file produced by an execution of a C program and reports the execution frequencies of each routine. It also reports the percentage of execution time spent in each routine.

prof normally reports times and frequencies spent for regions of programs between externally defined names. *progfile* is the executable program; if omitted, **l.out** is assumed. *monfile* is the monitor file produced during execution of the program; if omitted, **mon.out** is assumed.

To produce **mon.out**, a program must be compiled with the **-VPROF** option to **cc**. To profile all modules, each module must be compiled with this option.

The following options are available.

-a Profile all symbols, not just externals.

-b Print all bin information.

-c Print all call information.

-s Report stack usage high-water mark.

Files

l.out — Program file (with name list intact)
mon.out — Raw execution profile

See Also

cc, commands, ld, nm, profil

profile — System Maintenance

Set user's environment at login
/etc/profile

File **/etc/profile** holds a set of commands that the shell reads and executes when a user logs in.

If **/etc/passwd** specifies a program in the login shell slot, then **/etc/profile** is read by **/bin/sh** and those lines that begin with the command **export** are recognized as global environments, and the remainder of the line is inserted into the environment.

Please note that if **/bin/sh** is not the shell, any constructions other than

```
export foo=value
```

are not likely to work.

See Also

ksh, sh, system maintenance

prps — Command

Prepare files for PostScript-compatible printer
prps [*options*] [*file* ...]

prps invokes a driver for a PostScript-compatible device, typically a printer such as an Apple LaserWriter or a Hewlett-Packard LaserJet with a PostScript cartridge. It generates a PostScript program listing each input *file* and writes it to the standard output. If no *file* is given, **prps** reads the standard input.

The PostScript output program generates a sequence of standard 8.5 by 11-inch pages, each containing a header line (filename, current time and date, and page number) and a box that encloses the text of *file*. The default output typeface is ten-point Courier.

The most common use of **prps** is to print output via **/dev/hp**. For example, the command

```
prps file.c file.doc | hpr -B
```

pipes the output from **prps** into **hpr** (the print spooler for **/dev/hp**) to generate a listing of **file.c** and **file.doc**.

prps recognizes the following options:

-b Suppress the box around the page text. If the box is present, PostScript clips text that would extend beyond its right border.

-h Suppress the header line.

-i*n* Indent the left margin by an additional *n* characters.

-l Generate "landscape"-format output. **prps** normally generates output pages in "portrait" format (upright 8.5 by 11 inches). The **-l** option generates output pages in landscape format (11 by 8.5) instead. This option is useful for files with long lines; by default, it prints 46 lines per page.

-l2 Generate landscape-format output pages that each contain two side-by-side "pages" of text. This format is useful for saving paper, especially when used with a small size of type. As it prints in a small size of type, it prints 66 lines per page.

-n*name* Use *name* in place of the file name in the header line.

-t*N* Set tab stops at every *N* characters. The default tab setting is eight.

-ptsize Change the size of type to *ptsize* points. By default, **prps** sets its output in ten-point type. This yields 64 lines per normal output page, 46 lines in landscape format, and 52 lines per half page in **-l2** format. (Note that a "point" is one twelfth of a pica, which in turn is one sixth of an inch; thus, there are 72 points in an inch.) By specifying the *ptsize* on its command line, you can tell **prps** to use a different size of type. For example, **-8** tells **prps** to use eight-point type.

-p*N* Print *N* lines of text on each output page (or half page). Note that the point size determines how many lines fit on a page, and lines per page determine point size. If you specify both, **prps** will use the given values unless the lines do not fit at the given point size.

+*N* Skip the first *N* output pages.

Setting Fonts

prps recognizes the standard **nroff** font specification sequences and translates them into PostScript font specifications. The default font is Courier. Because the naming conventions for PostScipt fonts are anything but uniform, **prps** appends a suffix to the fontname to designate a Roman, boldface and italic font variety. The default suffix is '' for Roman, "-Bold" for bold and "-Oblique" for italic. These give the standard PostScript names for the Courier family, "Courier", "Courier-Bold", and "Courier-Oblique".

Option **-f***fontname* specifies an alternative *fontname*. Option **-Fs***Xsuffix* specifies an alternative font suffix, where *X* is one of the three characters **RBI** (for **R**oman, **B**old or **I**talic) and *suffix* is the desired suffix. For example, the option

```
-fTimes -FsR-Roman -FsI-Italic
```

generates the usual PostScript font names for the Times family, namely "Times-Roman", "Times-Bold", and "Times-Italic".

To spare you some of this grief, a few fonts have built-in abbreviations. Option **-F***X*, where *X* is one of the characters **ABHNPST**, specifies a PostScript fontname as follows:

-FA AvantGarde
-FB Bookman
-FH Helvetica
-FN Helvetica-Narrow
-FP Palatino
-FS New Century Schoolbook
-FT Times

These options also set each suffix appropriately for the desired font. However, font naming conventions may differ on various PostScript devices; examine the **prps** output and your device documentation if problems occur.

Examples

prps is especially useful as a way of printing the output of **nroff**, including manual pages. For example,

```
man prps | prps | hpr -B
```

or

```
man prps | prps -l2 | hpr -B
```

prints this Lexicon article in, respectively, portrait mode or two-page landscape mode. It looks nicer if you center the output with an indent:

```
man prps | prps -i8 | hpr -B
```

or

```
man prps | prps -l2 -i4 | hpr -B
```

See Also

commands, hp, hpr, pr, nroff, printer

ps — Command

Print process status

ps [-afglmnrtwx] [-c *sys* **] [-k** *mem* **]**

ps prints information about a process or processes. It prints the information in fields, followed by the command name and arguments. The fields include the following:

TTY The controlling terminal of the command, printed in short form. "44:" means **/dev/tty44** and "??:" means there is no controlling terminal.

PID Process id; necessary to know when the process is to be killed.

GROUP PID of the group leader of the process; the shell started up when the user logs in.

PPID PID of the parent of the process; very often a shell.

UID User id or name of the owner.

K Size of the process in kilobytes.

F Process flag bits, as follows:

PFCORE	00001	Process is in core
PFLOCK	00002	Process is locked in core
PFSWIO	00004	Swap I/O in progress
PFSWAP	00010	Process is swapped out
PFWAIT	00020	Process is stopped (not waited)
PFSTOP	00040	Process is stopped (waited on)
PFTRAC	00100	Process is being traced
PFKERN	00200	Kernel process
PFAUXM	00400	Auxiliary segments in memory
PFDISP	01000	Dispatch at earliest convenience
PFNDMP	02000	Command mode forbids dump
PFWAKE	04000	Wakeup requested

S State of the process, as follows:

R	Ready to run (waiting for CPU time)
S	Stopped for other reasons (I/O completion, pause, etc.)
T	Being traced by another process
W	Waiting for an existent child
Z	Zombie (dead, but parent not waiting)

EVENT The condition which the process is anticipating; not applicable if the process is ready to run.

CVAL SVAL IVAL RVAL
Scheduling information; bigger is better.

UTIME Time consumed while running in the program (in seconds).

STIME Time consumed while running in the system (in seconds).

Normally, **ps** displays the **TTY** and **PID** fields of each active process started on the caller's terminal, as well as the command name and arguments. The following flags can alter this

ps -agkutx

behavior.

a Display information about processes started from all terminals.

c The next argument *sys* gives the system executable image (default: **/coherent**). The namelist is searched for table addresses.

d Print information about status of loadable drivers.

f Blank fields have '-' place-holders. This enables field-oriented commands like **sort** and **awk** to process the output.

g Print the group leader field GROUP if the **l** option is given.

l Long format. In addition to the TTY and PID fields, prints the PPID, UID, K, F, S and EVENT fields.

k The next argument *mem* is the memory file (default: **/dev/mem**).

m Print the scheduling fields CVAL, SVAL, IVAL and RVAL.

n Suppress the header line.

r Print the real size of the process, which includes the user and auxiliary segments assigned to the process. Because the user segment (usually 1 kilobyte) is shared by all processes owned by that user, this may give a misleading total size for all the user's processes.

t Print elapsed CPU time fields UTIME and STIME.

w Wide format output; print 132 columns instead of 80.

x Display processes which do not have a controlling terminal (e.g. the swapper).

Files

/coherent — Default system file
/dev/mem — Default memory file
/dev/tty* — List of terminal names

See Also

commands, kill, mem, size, wait

Notes

Each process can modify or destroy its command name and arguments. The state of the system changes even as **ps** runs.

PS1 — Environmental Variable

User's default prompt
PS1=*prompt*

The environmental variable **PS1** sets the prompt for your shell. The default is **$**.

See Also

environmental variables, PS2, sh

PS2 — Environmental Variable

Prompt when user continues command onto additional lines
PS2=*prompt*

The environmental variable **PS2** sets the prompt that is displayed when a command extends onto additional input lines. The default is >.

See Also

environmental variables, PS1, sh

ptrace() — COHERENT System Call

Trace process execution
#include <signal.h>
int ptrace(*command, pid, location, value*)
int *command, pid*, ******location, value*;

ptrace provides a parent process with primitives to monitor and alter the execution of a child process. These primitives typically are used by a debugger such as **db**, which needs to examine and change memory, plant breakpoints, and single-step the child process being debugged.

Once a child process indicates it wishes to be traced, its parent issues various *commands* to control the child. *pid* identifies the affected process. The parent may issue a command only when the child process is in a stopped state, which occurs when the child encounters a signal. A special return value of 0177 from **wait** informs the parent that the child has entered the stopped state. The parent may then examine or change the child process memory space or restart the process at any point.

When the child process issues an **exec**, the child stops with signal **SIGTRAP** to enable the parent to plant breakpoints. The set user id and set group id modes are ineffective when a traced process performs an **exec.**

The following list describes each available *command*. A *command* ignores any arguments not mentioned.

0 This is the only *command* the child process may issue. It tells the system that the child wishes to be traced. Parent and child must agree that tracing should occur to achieve the desired effect. Only the *command* argument is significant.

1,2 The **int** at *location* is the return value. Command 1 signifies that *location* is in the instruction space, whereas command 2 signifies *data* space. Often these two spaces are equivalent.

3 The return value is the **int** of the process description, as defined in **sys/uproc.h.** This call may be used to obtain values such as hardware register contents and segment allocation information.

4,5 Modify the child process's memory by changing the **int** at *location* to *value*. Command 4 means instruction space and command 5 means data space. Shared segments may be written only if no other executing process is using them.

6 Modify the **int** at *location* in the process description area, as with command 3. The permissible values for *location* are restricted to such things as hardware registers and bits of machine status registers that the user may safely change.

7 This command restarts the stopped child process after it encounters a signal. The process resumes execution at *location*, or from where the process was stopped if *location* is **(int *)1.** *value* gives a signal number that the process receives as it restarts. This is normally the number of the signal that caused the process to stop, fetched from the process description area by a **3** command. If *value* is zero, the effect of the signal is ignored.

8 Force the child process to exit.

9 Like command **7**, except that the child stops again with signal **SIGTRAP** as soon as practicable after the execution of at least one instruction. The actual hardware method used to implement this command varies from machine to machine, explaining the imprecise nature of its definition. This call may provide part of the basis for breakpoints.

Files

<signal.h>
<sys/uproc.h>

See Also

db, COHERENT system calls, exec, signal(), wait()

Diagnostics

ptrace returns -1 if *pid* is not the process id of an eligible child process or if some other argument is invalid or out of bounds. Some commands may return an arbitrary data value, in which case **errno** should be checked to distinguish a return value of -1 from an error return.

Notes

There is no way to specify which signals should not stop the process.

pun — Definition

In the context of C, a **pun** occurs when a programmer uses one data form interchangeably with another. A pun most often occurs unintentionally when the programmer fails to declare a function as returning a pointer; by default, what the function returns is assumed to be an **int**, and is handled as such. No trouble will arise if the program is run on a machine that defines an **int** and a pointer to have the same length (e.g., i8086 SMALL model); however, such code cannot be transported to an environment in which this is not the case (e.g., i8086 LARGE model).

See Also

definitions, pointer, portability

pushd — Command

Push an item onto the directory stack
pushd *[directory0 ... directoryN]*

The COHERENT shell **sh** maintains an internal "directory stack", which is a stack of names of directories. You can manipulate this stack should you, for any reason, wish to traverse a number of directories quickly and efficiently.

The command **pushd** pushes *directory1* through *directoryN* onto the directory stack, and changes the current directory to the last directory pushed. If called without an argument, it transposes the last two directories on the directory stack.

See Also

commands, dirs, popd, sh

putc() — STDIO (stdio.h)

Write character into stream
#include <stdio.h>
int putc(*c*, *fp*) char *c*; FILE **fp*;

putc is a macro that writes a character *c* into the file stream pointed to by *fp*. It returns *c* upon success.

Example

The following example demonstrates **putc**. It opens an ASCII file and prints its contents on the screen. For another example of **putc**, see the entry for **getc**.

```
#include <stdio.h>
main()
{
     FILE *fp;
     int ch;
     int filename[20];

     printf("Enter file name: ");
     gets(filename);

     if ((fp = fopen(filename,"r")) != NULL) {
          while ((ch = fgetc(fp)) != EOF)
               putc(ch, stdout);
     } else
          printf("Cannot open %s.\n", filename);
     fclose(fp);
}
```

See Also

fputc(), getc(), putchar(), STDIO

Diagnostics

putc returns **EOF** when a write error occurs.

Notes

Because **putc** is a macro, arguments with side effects may not work as expected.

putchar() — STDIO (stdio.h)

Write a character onto the standard output
#include <stdio.h>
int putchar(*c*) char *c*;

putchar is a macro that expands to **putc(*c*, stdout)**. It writes a character onto the standard output.

Example

For an example of this routine, see the entry for **getchar**.

See Also

fputc(), putc(), STDIO

Diagnostics

putchar returns **EOF** when a write error occurs.

Notes

Because **putchar** is a macro, arguments with side effects may not work as expected.

puts() — STDIO (libc)

Write string onto standard output
#include <stdio.h>
int puts(*string*) **char** ******string*

puts appends a newline character to the string pointed to by *string*, and writes the result onto the standard output. **puts** returns a nonnegative value on success and **EOF** if a write error occurs.

Example

The following uses **puts** to write a string on the screen.

```
#include <stdio.h>

main()
{
     puts("This is a string.");
}
```

See Also

fputs(), STDIO

Notes

For historical reasons, **fputs** outputs the string unchanged, whereas **puts** appends a newline character.

putw() — STDIO (stdio.h)

Write word into stream
#include <stdio.h>
int putw(*word*, *fp*) **int** *word*; **FILE** ******fp*;

The macro **putw** writes *word* into the file stream pointed to by *fp*. It returns the value written.

putw differs from the related macro **putc** in that **putw** writes an **int**, whereas **putc** writes a **char** that is promoted to an **int**.

See Also

ferror(), STDIO

Diagnostics

putw returns **EOF** when an error occurs. You may need to call **ferror** to distinguish this value from a genuine end-of-file flag.

Notes

Because **putw** is a macro, arguments with side effects may not work as expected. The bytes of *word* are written in the natural byte order of the machine.

pwd — Command

Print the name of the current directory
pwd

pwd prints the name of the directory that you are in.

See Also

cd, commands, ksh, sh

Notes

Under the Korn shell, **pwd** is an alias for the expression **print -r $PWD**.

pwd.h — Header File

Declare password structure
#include <pwd.h>

The header file **pwd.h** declares the structure **passwd**, which is used to build COHERENT's password file. **passwd** is defined as follows:

```
struct passwd {
        char    *pw_name;     /* login user name */
        char    *pw_passwd;   /* login password */
        int     pw_uid;       /* login user id */
        int     pw_gid;       /* login group id */
        int     pw_quota;     /* file quota (unused) */
        char    *pw_comment;  /* comments (unused) */
        char    *pw_gecos;    /* (unused) */
        char    *pw_dir;      /* working directory */
        char    *pw_shell;    /* initial program */
};
```

For detailed descriptions of the above fields, see the entry for **passwd**.

See Also

endpwent(), getpwent(), getpwnam(), getpwuid(), header files, setpwent()

Q

qfind — Command

Find all files with a given name
qfind [-adp] *name* **...**
qfind [-b]

qfind prints the full path name of each file with a given *name*. When invoked with the **-b** option, it builds a data base in file **/usr/adm/qffiles**; this data base names every file and directory in the system. When invoked without the **-b** option, **qfind** reads this data base to find file names fairly quickly.

Normally, **qfind** prints the full path name of each file in the COHERENT system that ends with the given *name* (as they were at the time you last executed **qfind -b**.) With the **-d** option, **qfind** prints matching directories instead of files. With the **-a** option, **qfind** prints both matching files and matching directories.

Option **-p** specifies partial name matching. For example, **qfind -p foo** matches files **/src/foo.c** and **/doc/foo.r** as well as file **/usr/bin/foo.**

Files

/usr/adm/qffiles

See Also

commands, cron, find, whereis, which

Notes

Building the **qfind** data base with the **-b** option is slow, but it speeds finding files. You may find it convenient to use **cron** to execute **qfind -b** to rebuild the data base at night, or some other time when the machine is otherwise idle. The superuser **root** must run **qfind -b** if you want all files to appear in the data base.

qsort() — General Function (libc)

Sort arrays in memory
void qsort(*data*, *n*, *size*, *comp***) char ****data***; int** *n*, *size***; int (****comp***)();**

qsort is a generalized algorithm for sorting arrays of data in memory, using C. A. R. Hoare's "quicksort" algorithm. **qsort** works with a sequential array of memory called *data*, which is divided into *n* parts of *size* bytes each. In practice, *data* is usually an array of pointers or structures, and *size* is the **sizeof** the pointer or structure. Each routine compares pairs of items and exchanges them as required. The user-supplied routine to which *comp* points performs the comparison. It is called repeatedly, as follows:

```
(*comp)(p1, p2)
char *p1, *p2;
```

Here, *p1* and *p2* each point to a block of *size* bytes in the *data* array. In practice, they are usually pointers to pointers or pointers to structures. The comparison routine must return a negative, zero, or positive result, depending on whether *p1* is logically less than, equal to, or greater than *p2*, respectively.

Example

For an example of this function, see the entry for **malloc**.

See Also

general functions, shellsort(), strcmp(), strncmp()
The Art of Computer Programming, vol. 3

Notes

qsort differs from the other sorting function, **shellsort**, in that it uses a recursive algorithm that makes heavy use of the stack.

quot — Command

Summarize file-system usage
quot [-c] [-f] [-n] [-t] *filesystem*

quot produces several different summaries about the ownership of files for each *filesystem* argument given. When no options are specified, **quot** produces a two-column listing that gives the amount of space used by each user, sorted in decreasing order of file space used; the first column gives the number of blocks used and the second gives the use name. Space is always given in blocks.

Options are available to modify the normal output or specify a completely different action.

quot recognizes the following options:

-c Give a three-column breakdown of files by size. The first column contains all file sizes, in increasing order. The second column gives the number of files of the size indicated in the first. The third gives a cumulative sum of the sizes of all files less than or equal to the current size.

-f Add an initial column that contains the number of files to the front of the normal output.

-n Takes as input a list of i-numbers and file names, one per line and sorted in ascending order by i-number; ignore all lines not in this form. The output is in two columns: the first gives the owner and the second contains the file name for each entry in the output. This conforms to usage with the following pipeline:

```
ncheck filesystem | sort +0n | quot -n filesystem
```

-t To the normal output, add a line that contains totals.

quot runs much faster with a raw device for *filesystem*.

Only the superuser **root** can run **quot**.

Files

/etc/passwd

See Also

ac, commands, ncheck, sort

Notes

Sparse files are recorded as if they had all of their blocks allocated.

R

ram — Device Driver

Driver for manipulating RAM

The COHERENT **ram** devices let you allocate and use the random access memory (RAM) of the computer system directly. A typical use is for a RAM disk, which is a COHERENT file system kept in memory rather than on a floppy disk or hard disk.

The COHERENT RAM device driver has major number 8. It can be accessed either as a block-special device or as a character-special device. The high-order bit of the minor number gives a RAM device number (0 or 1), which lets you use up to two RAM devices simultaneously. The low-order seven bits specify the device size in 64-kilobyte increments. The first **open** call on a RAM device with nonzero size (1 to 127) allocates memory for the device; the system call **open** fails if sufficient memory is not available. Accessing a RAM device with a minor number specifying size zero frees the allocated memory, provided all earlier **open** calls have been closed.

Initially, COHERENT includes two block-special devices for RAM disks: the 512-kilobyte device **/dev/ram0** (8, 8) and the 192-kilobyte device **/dev/ram1** (8, 131). It also includes the devices **/dev/ram0close** (8, 0) and **/dev/ram1close** (8, 128). You should change the RAM devices to sizes appropriate for the amount of memory available on your system.

Examples

The following example formats and mounts a 512-kilobyte RAM disk on directory **/fast**.

```
mkdir /fast
/etc/mkfs /dev/ram0 1024
/etc/mount /dev/ram0 /fast
```

When the RAM disk is no longer needed, its allocated memory can be freed as follows:

```
/etc/umount /dev/ram0
cat /dev/null >/dev/rram0close
```

The next example replaces the default **/dev/ram0** with a one-megabyte device containing a COHERENT file system. The new minor number 16 specifies RAM device 0 and size 16 times 64 kilobytes (i.e., one megabyte). The new RAM device contains 2,048 blocks of 512 bytes each.

```
rm /dev/ram0
/etc/mknod /dev/ram0 b 8 16
/etc/mkfs /dev/ram0 2048
```

Files

/dev/ram*

See Also

compress, device drivers, fsck, mkfs, mount, umount, uncompress, zcat

Notes

Moving frequently used commands or files to a RAM disk can improve system performance substantially. However, the contents of a RAM device are lost if the system loses power, reboots, or crashes, files kept on a RAM disk should frequently be copied the hard disk or floppy disk.

If a RAM device uses most but not all available system memory, its **open** call will succeed but subsequent commands may fail because insufficient memory remains for the system.

The COHERENT installation program **/etc/build** uses RAM device **/dev/ram1** as a RAM disk during installation. Commands **compress**, **uncompress**, **zcat**, and **fsck** sometimes use **/dev/ram1** as a temporary storage device. Users should avoid using **/dev/ram1** as a RAM disk because of these programs. In addition, users of **compress**, **uncompress**, and **zcat** may have to change the size of **/dev/ram1** from the default size of 192 to 512 kilobytes, to handle files compressed to 16 bits. The following script makes this change; note that it must be run by the superuser **root**:

```
cat /dev/null >/dev/rram1close
rm /dev/ram1 /dev/rram1
mknod /dev/ram1  b 8 136
mknod /dev/rram1 c 8 136
```

Please note that increasing the size of **/dev/ram1** to 512 kilobytes requires a system with at least one megabyte of RAM.

rand() — General Function (libc)

Generate pseudo-random numbers
int rand()

rand generates a set of pseudo-random numbers. It returns integers in the range 0 to 32,767, and purportedly has a period of 232. **rand** will always return the same series of random numbers unless you first call the function **srand** to change **rand**'s *seed*, or beginning-point.

Example

This example demonstrates the functions **rand** and **srand**. It uses a threshold level that is passed in **argv[1]** (default, MAXVAL/2), the number of trials passed in **argv[2]** (default, 1,000), and a seed passed in **argv[3]** (default, no seeding).

```
#define MAXVAL 32767 /* range of rand: [0,2^15-1] */

main(argc, argv)
int argc; char *argv[];
{
	register int i, hits, threshold, ntrials;

	hits = 0;
	threshold = (argc > 1) ? atoi(argv[1]) : MAXVAL/2;
	ntrials = (argc > 2) ? atoi(argv[2]) : 1000;
	if (argc > 3)
		srand(atoi(argv[3]));

	for (i = 1; i <= ntrials; i++)
		if (rand() > threshold)
			++hits;

	printf("%d values above %d in %d trials (%D%%).\n",
		hits, threshold, ntrials, (100L*hits)/ntrials);
}
```

See Also

general functions, srand()
The Art of Computer Programming, vol. 2

random access — Definition

In the context of computing, **random access** means that an entity, such as memory, can be accessed at any point, not just at the beginning. This means that all points within memory can be accessed equally quickly. This contrasts with *sequential access*, in which entities must be accessed in a particular order, so that some entities take longer to access than do others.

A tape drive is an example of a sequential access device, i.e., the order in which data are read is dictated by the order in which they stream past the tape head. Random-access memory (RAM) is an example of random access. Hard disks and floppy disks combine elements of random access and sequential access.

RAM, which usually consists of semiconductor integrated circuits, is also strictly random access. In this regard, the term "RAM" is slightly misleading; a more accurate name would be "read/write memory", to contrast RAM with read-only memory (ROM), which is also random access memory.

See Also

definitions, read-only memory

ranlib — Definition

The **ranlib** is a "directory" that appears at the beginning of each library. It contains the name of each global symbol (i.e., function name) that appears within the library, and a pointer to the module in which that symbol is defined. Thus, the ranlib eliminates the need for the linker to search the entire library sequentially to find a given global symbol, which speeds up linking noticeably.

If the date on the library file is later than that in the ranlib header, the linker will ignore the ranlib and perform a sequential search through the library; the linker will also send the warning message

```
Outdated ranlib
```

to the standard error device. This is done to prevent the accidental use of an outdated ranlib, which could be disastrous. When you use the archiver **ar** to update a library or to create a new library, be sure to employ the options that update the ranlib as well as modify or create the library.

See Also

ar, date, ld, libraries, touch

ranlib — Command

Create index for object library
ranlib *library* ...

ranlib creates an index for a library created by the archiver **ar**. This greatly increases the speed with which the linker **ld** can search a library, and so link your program.

ranlib inserts at the beginning of the library an index of globals symbols that appear in the library. This list is in the order in which symbols appear in the library, to ensure that nothing links differently after creating a **ranlib** index. If the index already exists, **ranlib** updates it.

Files

_.SYMDEF — Index module

See Also

ar, ar.h, commands, ld

Diagnostics

ranlib issues appropriate messages for I/O errors or bad format files. It does not rewrite a library until the last possible moment, so the library is usually unchanged in case of error. **ranlib** processes each library independently. The exit status is the number of libraries in which errors were encountered.

rc — System Maintenance

Perform standard maintenance chores
/etc/rc

The shell script **/etc/rc** is executed by the **init** process when the COHERENT system enters multi-user mode. The commands in **rc** do such things as set the local time zone, and initialize the time **/usr/adm/wtmp**, which holds records of user logins.

See Also

brc, init, system maintenance

read() — COHERENT System Call (libc)

Read from a file
int read(*fd*, *buffer*, *n*) **int** *fd*; **char** **buffer*; **int** *n*;

read reads up to *n* bytes of data from the file descriptor *fd* and writes them into *buffer*. The amount of data actually read may be less than that requested if **read** detects **EOF**. The data are read beginning at the current seek position in the file, which was set by the most recently executed **read** or **lseek** routine. **read** advances the seek pointer by the number of characters read.

Example

For an example of how to use this function, see the entry for **open**.

See Also

COHERENT system calls, STDIO

Diagnostics

With a successful call, **read** returns the number of bytes read; thus, zero bytes signals the end of the file. It returns -1 if an error occurs, such as bad file descriptor, bad *buffer* address, or physical read error.

Notes

read is a low-level call that passes data directly to COHERENT. It should not be intermixed with high-level calls, such as **fread**, **fwrite**, or **fopen**.

read — Command

Assign values to shell variables
read *name* ...

read reads a line from the standard input. It assigns each token of the input to the corresponding shell variable *name*. If the input contains fewer tokens than the number of names specified, **read** assigns the null string to each extra variable. If the input contains more tokens than the number of names specified, **read** assigns the last *name* in the list the

remainder of the input.

The shell executes **read** directly.

Example

The command

```
read foo bar baz
hello how are you
```

parses the line "hello how are you" and assigns the tokens to, respectively, the shell variables **foo**, **bar**, and **baz**. If you further type

```
echo $foo
echo $bar
echo $baz
```

you will see:

```
hello
how
are you
```

See Also

commands, ksh, sh

Diagnostics

read normally returns an exit status of zero. If it encounters end of file or is interrupted while reading the standard input, it then returns one.

readonly — C Keyword

Storage class

readonly is a C keyword that modifies data declarations. As its name implies, the **readonly** modifier declares that data are to be read only; this helps protect key data against casual modification by the user or another programmer.

See Also

C keywords, keyword

Notes

The ANSI standard for the C language eliminates this keyword.

readonly — Command

Mark a shell variable as read only
readonly

Mark each *variable* as a read-only shell variable. The shell will not permit subsequent assignments to a readonly variable. With no arguments, **readonly** prints the name and value of each read-only variable.

See Also

commands, ksh, sh

read-only memory — Definition

As its name suggests, **read-only memory**, or ROM, is memory that can be read but not overwritten. It most often is used to store material that is used frequently or in key situations, such as a language interpreter or a boot routine.

See Also

definitions, random access

realloc() — General Function (libc)

Reallocate dynamic memory
char *realloc(*ptr***, *** size***) char *** ptr***; unsigned*** size***;**

realloc helps you manage a program's arena. It returns a block of *size* bytes that holds the contents of the old block, up to the smaller of the old and new sizes. **realloc** tries to return the same block, truncated or extended; if *size* is smaller than the size of the old block, **realloc** will return the same *ptr*.

If *ptr* is set to NULL, **realloc** behaves like **malloc**.

Example

For an example of this function, see the entry for **calloc**.

See Also

arena, calloc(), free(), general functions, malloc(), memok(), setbuf()

Diagnostics

realloc returns NULL if insufficient memory is available. It prints a message and calls **abort** if it discovers that the arena has been corrupted, which most often occurs by storing past the bounds of an allocated block. **realloc** will behave unpredictably if handed an incorrect *ptr*.

reboot — Command

Reboot the COHERENT system
/etc/reboot [-p]

reboot reboots the COHERENT system. The option **-p** prompts the user if she really wishes to reboot before executing the reboot.

reboot can be run only by the superuser.

The COHERENT system should be rebooted only while in single-user mode. Failure to return to single-user mode before rebooting could damage the COHERENT file system and destroy data.

See Also

commands, shutdown

ref — Command

Display a C function header
ref *function*

ref looks up the function header of *function* in any of a series of reference files built by the command **ctags**. It is used by the **elvis** editor's **<shift-K>** command. This command checks the file **refs** in the current directory.

See Also

commands, ctags, elvis

Notes

ref is copyright © 1990 by Steve Kirkendall, and was written by Steve Kirkendall (kirkenda@cs.pdx.edu or ...uunet!tektronix!psueea!eecs!kirkenda), assisted by numerous volunteers. It is freely redistributable, subject to the restrictions noted in included documentation. Source code for **ref** is available through the Mark Williams bulletin board, USENET, and numerous other outlets.

Please note that **ref** is included as a service to COHERENT users, but is not supported by Mark Williams Company. *Caveat utilitor.*

register — C Keyword

Storage class

register is a C keyword that declares a class of data storage. A variable so declared may be stored in a register, which may increase the speed with which it is read by a program.

See also

auto, C keywords, extern, register variable, static

register variable — Definition

register is a C storage class. A **register** declaration tells the compiler to try to keep the defined local data item in a machine register. Under the COHERENT C compiler, the **int foo** can be declared to be a register variable with the following statement:

```
register int foo;
```

COHERENT places the first two register variables declared in a function into registers SI and DI if the variable type is appropriate, i.e., **int** or SMALL-model pointer. Subsequent **register** declarations are ignored, because no registers are left to hold them. Note because of this fact, declaring more than two register variables may slow processing rather than speed it.

By definition of the C language, registers have no addresses, so you cannot pass the address of register variable as an argument to a function. For example, the following code will generate an error message when compiled:

```
register int i;
  . . .
dosomething(&i);      /* WRONG */
```

This rule applies whether or not the variable is actually kept in a register.

Placing heavily-used local variables into registers often improves performance, but in some cases declaring **register** variables can degrade performance somewhat.

See Also

auto, definitions, extern, static, storage class

rename — Technical Information

How to rename a file

The COHERENT system has no "rename" procedure *per se*. On the shell level, you can use the **mv** command to rename a file. To rename a file from within a C program, you must use the COHERENT system calls **link** and **unlink**.

Example

The following program demonstrates how to use **link** and **unlink** to rename a file.

```
#include <stdio.h>
main(argc, argv) int argc; char *argv[];
{
	register char *old, *new;

	if (argc != 3) {
		fprintf(stderr, "Usage: rename old new\n");
		exit(1);
	}
	old = argv[1];
	new = argv[2];

	if (link(old, new) == -1) {
		fprintf(stderr, "rename: link(%s, %s) failed\n", old, new);
		exit(1);
	} else if (unlink(old) == -1) {
		fprintf(stderr, "rename: unlink(%s) failed\n", old);
		exit(1);
	}
	exit(0);
}
```

See Also

mv, technical information

restor — Command

Restore file system

restor *command* [*dump_device*] [*filesystem*] [*file* ...]

restor copies to the hard disk one or more files from floppy disks or tapes written by the command **dump**.

restor recognizes the following *commands*:

r Mass restore both full and incremental dump disks/tapes into the *filesystem*. The target file system must have enough data blocks and i-nodes to hold the dump.

The mass restoration is performed in three phases. In phase 1, **restor** clears all i-nodes that were either clear at dump time or are going to be restored. Any allocated blocks are released. Second, it restores all files on the disk. The i-numbering is preserved; however, data blocks are allocated in the standard fashion. Third, a pass is made over the i-nodes and the list of free i-nodes in the superblock is updated.

Restoration begins immediately, using the currently mounted disk or tape.

R Like the **r** command, except that it pauses to ask for numbers of disks or reels.

t Read the header from the dump. Display the date the dump was written and the "dump since" date that produced the dump.

x Extract each *file* from the dump and restore it to the hard disk. All file names are absolute path names starting at the root of the dump (the first directory dumped, which is always the root directory of the file system). A numeric file name is taken to be an i-number on the dumped file system, permitting restore by i-number.

restor looks up each *argument* file in the directories of the dumped file system and prints out each name and associated i-number. **restor** extracts the files from the currently mounted dump disk or tape, and writes the extracted files into the current directory. Extracted files are named after their i-numbers.

X Like command x, except that before it begins, it asks you for the number of the disk (or the reel number of the dump tape). It continues asking for dump disks until all files have been extracted or you types **<ctrl-D>**.

Each of the above commands can be modified either or both of the following modifiers:

f Tell **restor** to take the next *argument* as the path name of the dump device (floppy-disk drive or tape drive). If the **f** modifier is not specified, **restor** uses the device **/dev/dump**.

v Verbose output. Tell **restor** to print a step-by-step trace of its actions when restoring an entire file system. This is for discovering what went wrong when a mass restore runs into trouble.

Restoring from a Damaged Medium

As noted below, **dump** requires that its output be written to disks or tapes that are free of defects. Restoring a file system from a damaged medium is difficult and is not associated with a high probability of success; if, however, you must try to do so, the following directions will give you a fighting chance of restoring your data.

1. Use the command **fdformat** to format a blank disk. Use the command **badscan** to examine it for bad sectors; if it does have bad sectors, put it aside and try another.
2. Use the command **dd** to copy the bad disk to directory **/tmp***foo1* **dd** should die at the bad sector in the disk.
3. **dd** again to directory **/tmp/***foo2* using that command's **skip=***n* to skip past the bad sector (or sectors).
4. Repeat step 3 (if it died too) until the end of the disk is reached. Now you have a set of directories named **/tmp/***foo[1...n]* that contain parts of the bad disk.
5. Use the command

   ```
   dd if=/tmp/foo1 of=/dev/fha0
   ```

   with the new, defect-free disk.
6. Now, use the command

   ```
   dd if=/tmp/foo2 of=/dev/fha0 seek=whatever
   ```

   to place *foo2* into the right place on the new disk.
7. Repeat 6 for each directory *foo3* through *fooN*.
8. Create a 512-byte file that contain the string

   ```
   GARBAGE\n
   ```

   repeated 64 times. Use **dd** to copy it into new disk where the bad sectors were.

Now, you *should* have a disk that is a mirror image of the old, damaged dump disk. Each bad sectors will have been replaced by 64 iterations of the string **GARBAGE\n**. As noted, there is no guarantee that this scheme will work in every instance (the chances of error are quite high), but it will give you a fighting chance to save your data.

Files

/dev/dump — Dump device
/etc/ddate — Dump date file

See Also

commands, dump, dumpdir

Diagnostics

Most of the diagnostics produced by **restor** are self-explanatory, and are caused by internal table overflows or I/O errors on the dump medium or file system.

If the dump spans multiple disks or reels, **restor** asks you to mount the next disk at the appropriate time. Type a newline when the disk has been mounted. **restor** verifies that this is the correct disk, and gives you another chance if the disk number in the dump header is incorrect.

Notes

You cannot perform a mass restore onto a live root partition. Instead, boot a stand-alone version of COHERENT on a floppy-disk drive, or boot from an alternative COHERENT file system on another hard-disk partition before you attempt to do a mass restoration.

The handling of tapes with multiple dumps on them (created by dumping to the no rewind special files) is not very general. Basically, **restor** assumes that tapes holding multiple dumps and tapes holding dumps that span multiple reels are mutually exclusive. You can restore from any file on a reel by positioning the tape and then restoring with the **x** or **r** commands, which do not reposition the tape. It is (almost) impossible to use the **X** or **R** commands, as the position of the dump tape will be lost when **restor** closes it.

dump requires that its output be written to disks that are free of bad sectors. If you write a dump to a disk with bad sectors, you will not be able to restore files from that disk. See **dump** for directions on processing disks to ensure that they are free of bad sectors.

return — C Keyword

Return a value and control to calling function

return is a C statement that returns a value from a function to the function that called it. **return** can be used without a value, to return control of the program to the calling function; also, the calling function is free to ignore the value **return** hands it. Note that it is good programming practice to declare functions that return nothing to be of type **void**.

A function can return only one value to the function that called it. Most often, this value is used to signal whether the function performed successfully or not.

See Also

C keywords

rev — Command

Print text backwards
rev [*file* ...]

rev reverses the order of the characters in each line of each input *file* and writes the result to the standard output. If no *file* is specified, the standard input is used instead.

Example

The following allows you to give a command like Mandrake the Magician

```
rev
Rocks break down wall!
<ctrl-D>
```

which displays:

```
!llaw nwod kaerb skcoR
```

on your screen.

See Also

commands

rewind() — STDIO Function (libc)

Reset file pointer

#include <stdio.h>
int rewind(*fp*) FILE **fp*;

rewind resets the file pointer to the beginning of stream *fp*. It is a synonym for **fseek(*fp*, 0L, 0)**.

Example

For an example of this routine, see the entry for **fscanf**.

See Also

fseek(), ftell(), lseek(), STDIO

Diagnostics

rewind returns **EOF** if an error occurs; otherwise, it returns zero.

rindex() — String Function (libc)

Find a character in a string

char *rindex(*string*, *c*) char **string*; char *c*;

rindex scans *string* for the last occurrence of character *c*. If *c* is found, **rindex** returns a pointer to it. If it is not found, **rindex** returns NULL.

Example

This example uses **rindex** to help strip a sample file name of the path information.

```
#include <stdio.h>
#define PATHSEP '/'        /* path name separator */

extern char *rindex();
extern char *basename();

main()
{
     char *testpath = "/foo/bar/baz";

     printf("Before massaging: %s\n", testpath);
     printf("After massaging: %s\n", basename(testpath));
}
```

```
char *basename(path)
char *path;
{
     char *cp;
     return (((cp = rindex(path, PATHSEP)) == NULL)
           ? path : ++cp);
}
```

See Also

index(), string functions

Notes

This function is identical to the function **strrchr**, which is described in the ANSI standard.

COHERENT includes **strrchr** in its libraries. It is recommended that it be used instead of **rindex** so that programs more closely approach strict conformity with the ANSI standard.

rm — Command

Remove files
rm [**-firtv**] *file* ...

rm removes each *file*. If no other links exist, **rm** frees the data blocks associated with the file.

To remove a file, a user must have write and execute permission on the directory in which the file resides, and must also have write permission on the file itself. The force option **-f** forces the file to be removed if the user does not have write permission on the file itself. It suppresses all error messages and prompts.

The interactive option **-i** tells **rm** to prompt for permission to delete each *file*.

The recursive removal option **-r** causes **rm** to descend into every directory, search for and delete files, and descend further into subdirectories. Directories are removed if the directory is empty, is not the current directory, and is not the root directory.

The test option **-t** performs all access testing but removes no files.

The verbose option **-v** tells **rm** to print each file rm and the action taken. In conjunction with the **-t** option, this allows the extent of possible damage to be previewed.

See Also

commands, ln, rmdir

Notes

Absence of delete permission in parent directories is reported with the message "*file*: permission denied". Write protection is not inherited by subdirectories; they must be protected individually.

Note that unlike the similarly named command under MS-DOS, COHERENT's version of **rm** will *not* prompt you if you request a mass deletion. Thus, the command

```
rm *
```

will silently and immediately delete all files in the current directory. *Caveat utilitor!*

rmdir — Command

Remove directories
rmdir [**-f**] *directory* ...

rmdir removes each *directory*. This will not be allowed if a *directory* is the current working directory or is not empty. The force option **-f** allows the superuser to override these restrictions. **rmdir** removes the '.' and '..' entries automatically. Note that using the **-f** option on a directory that is not empty will damage the file system, and require that it be fixed with **fsck**.

See Also

commands, mkdir, rm

Notes

rmdir -f does *not* remove files from a nonempty directory; it simply orphans them. To remove a nonempty directory and its contents, use **rm -r** instead.

root — Definition

root is the login name for the superuser.

See Also

definitions, superuser

rpow() — Multiple-Precision Mathematics

Raise multiple-precision integer to power
#include <mprec.h>
void rpow(*a, b, c*)
mint **a, *b, *c;

The COHERENT system includes a suite of routines that allow you to perform multiple-precision mathematics. The function **rpow** sets the multiple-precision integer (or **mint**) pointed to by *c* to the value pointed to by *a* raised to power of the value pointed to by *b*.

See Also

multiple-precision mathematics

RS-232 — Technical Information

COM port wiring

This article details the connections (pinouts) of EIA standard RS-232C. This connect consists of a D-shaped plug with 25 pins in two rows: 13 pins in the upper row and 12 in the lower. This interface is commonly used by devices that require a serial interface to a computer; these devices include modems, terminals, serial printers, and such specialized devices as bar-code scanners. In addition, this articles gives the pinouts of the nine-pin DB-9P connector, which is a nine-pin version of the RS-232 that is commonly used in AT and AT-compatible computers.

RS-232 Pinout

The following table gives the 25-pin EIA standard RS-232C pinouts. It also gives:

- Nine-pin DB-9P convention
- Common abbreviations of signal names
- Abbreviations of RS-232 signal names
- Equivalent CCITT signal-number designations
- Signal direction (as appropriate)
- Signal description

Please note that in this table, **DTE** stands for "data terminal equipment" and refers to terminal-type equipment such as a PC or a terminal, whereas **DCE** stands for "data communications equipment" and refers to modems and modem-type equipment.

DB-25 Pin #	*DB-9 Pin #*	*Common Name*	*EIA*	*CCITT*	*DTE-DCE*	*Description*
1		FG	AA	101	—	Frame ground
2	3	TD	BA	103	→	Transmitted data
3	2	RD	BB	104	←	Received data
4	7	RTS	CA	105	→	Request to send
5	8	CTS	CB	106	←	Clear to send
6	6	DSR	CC	107	←	Data set ready
7	5	SG	AB	102	—	Signal ground
8	1	DCD	CF	109	←	Data carrier detect
9		—	—	—	—	Positive DC test voltage
10		—	—	—	—	Negative DC test voltage
11		QM	—	—	←	Equalizer mode
12		SDCD	SCF	122	←	Secondary carrier detect
13		SCTS	SCB	121	←	Secondary clear to send
14		STD	SBA	118	→	Secondary transmitted data
15		TC	DB	114	←	Transmitter clock
16		SRD	SBB	119	←	Secondary receiver clock
17		RC	DD	115	→	Receiver clock
18		DCR	—	—	←	Divided clock receiver
19		SRTS	SCA	120	→	Secondary request to send
20	4	DTR	CD	108.2	→	Data terminal ready
21		SQ	CG	110	←	Signal quality
22	9	RI	CE	125	←	Ring indicator
23		—	CH	111	→	Data rate selector
24		TC	DA	113	←	Transmitted clock
25						

Files

/usr/pub/rs232 — On-line version of above table

See Also

modem control, technical information, terminal

Seyer, M.D.: *RS-232 Made Easy: Connecting Computers, Printers, Terminals, and Modems.* Englewood Cliffs, NJ, Prentice-Hall Inc., 1984.

Notes

Serial ports on the back of the PC use either a 25-pin male (DB-25P) or a nine-pin male (DB-9P) connector. Due to what can only be considered as extreme stupidity, the 25-pin female (DB-25S) connector was chosen for the parallel printer port, rather than using the usual 36-pin parallel connector. Do not confuse these ports when wiring custom cable assemblies, as you can damage your equipment!

rvalue — Definition

An **rvalue** is the value of an expression. The name comes from the assignment expression **e1=e2;**, in which the right operand is an rvalue.

Unlike an lvalue, an rvalue can be either a variable or a constant.

See Also

definitions, lvalue

S

sa — Command

Print a summary of process accounting
sa [-abcjlmnrstu] [-v *N*] [*file*]

One of the accounting mechanisms available on the COHERENT system is *process accounting* (also called *shell accounting*), which records the commands executed by each user. The command **accton** enables or disables shell accounting.

The command **sa** scans the accounting information in *file* and prints a summary. If *file* is omitted, it reads the file **/usr/adm/acct** by default. For each command executed, **sa** prints the number of calls made, the total CPU time (user and system), and the total real time. The output is ordered by decreasing CPU time.

sa recognizes the following options:

a Place commands executed only once and command names with unprintable characters in the category "***other".

b Sort by average CPU time per call.

c Also print CPU time as a percentage of all CPU time used.

j Print average times per call rather than totals.

l Separate user and system time.

m Accumulate information for each user rather for each command.

n Sort by number of calls.

r Reverse the order of the sort.

s After scanning, condense the accounting file and merge it into the summary files.

t Also print the CPU time as a percentage of real time.

u Print the user and command for each accounting record; this option overrides all others.

v *N* For commands called no more than *N* times, where *N* is a digit, **sa** asks whether to place the command in the category "**junk**".

sa uses the summary files **/usr/adm/savacct** and **/usr/adm/usracct** to lessen disk usage.

Files

/usr/adm/acct — Default account data
/usr/adm/savacct — Summary
/usr/adm/usracct — Summary

See Also

ac, acct(), acct.h, accton, commands

Notes

The file **/usr/adm/acct** can become very large; therefore, you should truncate it periodically. Special care should be taken if process accounting is enabled on a COHERENT system with limited disk space.

sbrk() — COHERENT System Call (libc)

Increase a program's data space
char *sbrk(*increment***) unsigned int** *increment***;**

sbrk increases a program's data space by *increment* bytes. It increments the variable **__end**; this variable is set by the C runtime startup routine, and points to the end of the program's data space. The memory allocation routine **malloc** calls **sbrk** should you attempt to allocate more space than is available in the program's data space.

sbrk returns a pointer to the previous setting of **__end** if the requested memory is available, or NULL if it is not.

See Also

brk(), COHERENT system calls, malloc()

Notes

sbrk will not increase the size of the program data area if the physical memory requested exceeds the physical memory allocated by COHERENT, or if the requested memory exceeds the limit set in the user-defined variable **maxmem**. **sbrk** does not keep track of how space is used; therefore, memory seized with **sbrk** cannot be freed. *Caveat utilitor.*

scanf() — STDIO (libc)

Accept and format input
#include <stdio.h>
int scanf(*format, arg1, ... argN***)**
char *****format*; **[data type] ****arg1*, ... ****argN***;

scanf reads the standard input, and uses the string *format* to specify a format for each *arg1* through *argN*, each of which must be a pointer.

scanf reads one character at a time from *format*; white space characters are ignored. The percent sign character '%' marks the beginning of a conversion specification. '%' may be followed by characters that indicate the width of the input field and the type of conversion to be done.

scanf reads the standard input until the return key is pressed. Inappropriate characters are thrown away; e.g., it will not try to write an alphabetic character into an **int**.

The following modifiers can be used within the conversion string:

1. The asterisk '*', which indicates that the next input field should be skipped rather than assigned to the next *arg*.
2. A string of decimal digits, which specifies a maximum field width.
3. An **l**, which specifies that the next input item is a **long** object rather than an **int** object. Capitalizing the conversion character has the same effect.

The following conversion specifiers are recognized:

c Assign the next input character to the next *arg*, which should be of type **char ***.

d Assign the decimal integer from the next input field to the next *arg*, which should be of type **int ***.

D Assign the decimal integer from the next input field to the next *arg*, which should be of type **long ***.

e Assign the floating point number from the next input field to the next *arg*, which should be of type **float** *.

E Assign the floating point number from the next input field to the next *arg*, which should be of type **double** *.

f Same as **e**.

F Same as **E**.

o Assign the octal integer from the next input field to the next *arg*, which should be of type **int** *.

O Assign the octal integer from the next input field to the next *arg*, which should be of type **long** *.

r The next argument points to an array of new arguments that may be used recursively. The first argument of the list is a **char** * that contains a new format string. When the list is exhausted, the routine continues from where it left off in the original format string.

s Assign the string from the next input field to the next *arg*, which should be of type **char** *. The array to which the **char** * points should be long enough to accept the string and a terminating null character.

x Assign the hexadecimal integer from the next input field to the next *arg*, which should be of type **int** *.

X Assign the hexadecimal integer from the next input field to the next *arg*, which should be of type **long** *.

It is important to remember that **scanf** reads up, but not through, the newline character: the newline remains in the standard input device's buffer until you dispose of it somehow. Programmers have been known to forget to empty the buffer before calling **scanf** a second time, which leads to unexpected results.

Example

The following example uses **scanf** in a brief dialogue with the user.

```
#include <stdio.h>

main()
{
     int left, right;

     printf("No. of fingers on your left hand:  ");
     /* force message to appear on screen */
     fflush(stdout);
     scanf("%d", &left);

     /* eat newline char */
     while(getchar() != '\n')
          ;

     printf("No. of fingers on your right hand:  ");
     fflush(stdout);
     scanf("%d", &right);
```

```
        /* again, eat newline */
        while(getchar() != '\n')
                ;

        printf("You've %d left fingers, %d right, & %d total\n",
                left, right, left+right);
}
```

See Also

fscanf(), sscanf(), STDIO

Diagnostics

scanf returns the number of arguments filled. It returns **EOF** if no arguments can be filled or if an error occurs.

Notes

Because C does not perform type checking, it is essential that an argument match its specification. For that reason, **scanf** is best used to process only data that you are certain are in the correct data format. The use of upper-case format characters to specify long arguments is not standard; use the 'l' modifier for portability.

scanf is difficult to use correctly, and its misuse can be associated with intermittent and dangerous bugs. Rather than use **scanf** to obtain a string from the keyboard: it is recommended that you use **gets** to obtain the string, and use **strtok** or **sscanf** to parse it.

scat — Command

Print text files one screenful at a time
scat [[*option* ...] [*file* ...]] ...

scat prints each *file* on the standard output, one screenful (24 lines) at a time if the output is a screen. **scat** reads and prints the standard inputif no *file* is named.

The text is processed to allow convenient viewing on a screen; the options described below select the nature of the processing. Options begin with '-' and may be interspersed with file names.

scat scans two argument lists. The first is in the environmental **SCAT**. It should consist of arguments separated by white space (space, tab, or newline characters), with no quoting or shell metacharacters. This string is a useful place to set terminal-dependent parameters (such as page width and length) and to place invocation lists (see below). The second argument list is supplied on the command line.

scat recognizes the following options:

-l Do not stop at EOF if exactly one file was specified on the command line.

-b*n* Begin output at input line *n*.

-c Represent all control characters unambiguously. With this option, **scat** prints control characters in the range 0-037 as a character in the range 0100-0137 prefixed by a carat '^'; for example, SOH appears as "^A" and DEL as "^?" It prints mark-parity characters (in the range of 0200-0377) with '~'; for example, mark-parity 'A' and SOH appear as "~A" and "~^A", respectively. It also prefixes the characters '^', '~', and '\' with a '\'. This option overrides the option **-t**.

-cs Like **-c**, but map space ' ' to underscore '_' and prefix underscore '_' with '\'.

-ct Like **-c**, but map tabs to spaces, not "^I".

-i*n* Shift the display window right *n* columns into the text field. This is useful for viewing long lines.

-l*n* Set the display window length to *n* lines. The default is 24 normally, 34 for the Tek 4012.

-n Number input lines; wrapped lines are not numbered.

-r Remote; the output is not paged.

-s Skip empty lines.

-S*n* Seek *n* bytes into input before processing.

-t Truncate long lines. Normally, **scat** wraps each long line, with the interrupted portion delimited by a '\'.

-w*n* Set the display window width to *n* columns. The default is 80 normally, 72 for the Tek 4012.

-x Expand tabs.

-. *suffix* Invoke options by file-name suffix. If a file name ends with *.suffix*, then **scat** scans the argument sublist starting immediately after the invocation flag. New options will apply to the invoking file only. A sublist is terminated by the end of the argument list, by a file name, by the "--" flag, or by another "-." (invocation lists do not nest).

-- Terminate a sublist (see previous option).

Numbers may begin with **0** to indicate octal, and may end in **b** or **k** to be scaled by 512 or 1,024, respectively.

If the output is being paged, **scat** waits for a user response, which may be one of the following:

newline	Display next page
/	Display next half-page
space	Display next line
f	Print current file name and line number
n	**scat** next file
q	Quit

Example

The following shows how to use the environment argument list, invocation lists, and sublists:

```
SCAT="-l24 -.c -n -.s -b5"
export SCAT
scat *.c *.s
```

After processing the **SCAT** argument list, **scat** processes the command line argument list "***.c *.s**" with the page length at 24 lines. If a file is a C source ("***.c**") the invoke option in the **SCAT** argument list numbers the output lines. If a file is an assembly source ("***.s**") **scat** skips the first four lines.

See Also

cat, commands, pr

sched.h — Header File

Define constants used with scheduling
#include <sys/sched.h>

sched.h defines constants and structures that are used by routines that perform scheduling.

See Also

header files

SCSI — Device Driver

SCSI device drivers

The COHERENT SCSI series of device drivers lets you use SCSI-interface devices attached to host adapters from several vendors.

All COHERENT SCSI device drivers use major number 13, thus allowing all SCSI devices to be accessed via standard device-naming conventions. Peripherals can be accessed as either block- or character-special devices. The minor number specifies the device and partition number for disk-type devices; this allows the use of up to eight SCSI identifiers (SCSI-ID's), with up to four logical unit numbers (LUNs) per SCSI-ID and up to four partitions per LUN. Tape and other special devices decode the minor number to perform special operations such as "rewind on close" or "no rewind on close".

The first **open** call on a SCSI disk device allocates memory for the partition table and reads it into memory.

See the release notes for further information regarding supported host adapters and peripherals.

Files

/dev/sd* — block-special devices
/dev/rsd* — character-special devices

See Also

aha154x, device drivers, drvld, ss

Notes

The Mark Williams Company's bulletin board makes available loadable device drivers for various SCSI host adapters, as well as device driver updates. See the release notes for further information.

sdiv() — Multiple-Precision Mathematics

Divide multiple-precision integers
#include <mprec.h>
void sdiv(*a, n, q, ip*)
mint **a*, **q*; **int** *n*, **ip*;

The COHERENT system includes a suite of routines that allow you to perform multiple-precision mathematics. The function **sdiv** divides the multiple-precision integer (or **mint**) pointed to by *a* with the integer *n*, which is in the range 1 <= *n* <= 128. It writes the quotient into the **mint** pointed to by *q* and the remainder into the integer pointed to by *ip*.

See Also

multiple-precision mathematics

SECONDS — Environmental Variable

Number of seconds since current shell started

The Korn shell stores in environmental variable **SECONDS** the number of seconds since the current shell was started.

See Also

environmental variables, ksh

security — Technical Information

Because COHERENT is a multi-user, multi-tasking operating system which can support users from remote terminals, steps must be taken to ensure that the system is secure. Sensitive information that is stored on the system must be protected from being read or copied by unauthorized persons; files must be protected against vandalization by intruders. Unless a reasonable degree can be guaranteed, no multi-user operating system can be trusted to archive important information.

In one sense, it is easy to achieve perfect security in a computer system. As Grampp and Morris have noted, "It is easy to run a secure computer system. You merely disconnect all dial-up connections, put the machine and its terminals in a shielded room, and post a guard at the door." For practical uses, however, security means balancing ease of access against restrictiveness: users should have easy access to what is properly theirs, and should be barred from system facilities that do not belong to them.

The COHERENT system has the following tools to assist with security.

Passwords

Every user account can be "locked" with a password. Each user can assign her own password, and the system administrator can set passwords for the superusers **root** and **bin**.

Passwords should be changed frequently. A password should have at least six characters, should *not* be a common name or word, and preferably should include a mixture of upper- and lower-case letters, to prevent decryption by brute-force methods.

Passwords should be guarded jealously. In particular, the password for the superuser **root** should be kept secret, as she can read every file and execute every program throughout the system.

Permissions

Execution of system-level programs, such as **mount**, is restricted to the superuser **root**. This prevents intruders from seizing superuser permissions through unauthorized manipulation of system services. Ordinary users are also restricted from directly access system devices, for the same reason.

Encryption

The command **crypt** performs rotary encryption, similar to that used by the German Enigma machine. Files of sensitive information should be encrypted, to protect them against being read by unauthorized persons. Note that encryption is the only true defense against unauthorized reading: not even the superuser can read an encrypted file unless she has the encryption key.

Many COHERENT systems have only one user and are not networked; for such installations, the normal level of security may be an annoyance. Passwords can be turned off by using the command **passwd** to set the password to **<return>**. The command **chmod**

can be used to widen access to devices and system-level utilities; see the Lexicon entry for **chmod** for more information on file access.

Security ultimately is a system-wide responsibility. To quote Grampp and Morris, "By far, the greatest security hazard for a system ... is the set of people who use it. If the people who use a machine are naive about security issues, the machine will be vulnerable regardless of what is done by the local management. This applies particularly to the system's administrators, but ordinary users should also take heed."

See Also

chmod, crypt, passwd, technical information

Grampp, F.T., Morris, R.H.: UNIX operating system security. *AT&T Bell Lab Tech J* 1984;8:1649-1672.

sed — Command

Stream editor

sed [-n] [-e *commands*] **[-f** *script*] ... *file* ...

sed is a non-interactive text editor. It reads input from each *file*, or from the standard input if no file is named. It edits the input according to commands given in the *commands* argument and the *script* files. It then writes the edited text onto the standard output.

sed resembles the interactive editor **ed**, but its operation is fundamentally different. **sed** normally edits one line at a time, so it may be used to edit very large files. After it constructs a list of commands from its *commands* and *script* arguments, **sed** reads the input one line at a time into a *work area*. Then **sed** executes each command that applies to the line, as explained below. Finally, it copies the work area to the standard output (unless the **-n** option is specified), erases the work area, and reads the next input line.

Line Identifiers

sed identifies input lines by integer line numbers, beginning with one for the first line of the first *file* and continuing through each successive *file*. The following special forms identify lines:

n A decimal number *n* addresses the *n*th line of the input.

. A period '.' addresses the current input line.

**$** A dollar sign '$' addresses the last line of input.

/pattern/
A *pattern* enclosed within slashes addresses the next input line that contains *pattern*. Patterns, also called *regular expressions*, are described in detail below.

Commands

Each command must be on a separate line. Most commands may be optionally preceded by a line identifier (abbreviated as *[n]* in the command summary below) or by two-line identifiers separated by a comma (abbreviated as *[n[,m]]*). If no line identifier precedes a command, **sed** applies the command to every input line. If one line identifier precedes a command, **sed** applies the command to each input line selected by the identifier. If two-line identifiers precede a command, **sed** begins to apply the command when an input line is selected by the first, and continues applying it through an input line selected by the second.

sed recognizes the following commands:

*[n]***=** Output the current input line number.

[n[,m]]!*command*
Apply *command* to each line *not* identified by *[n[,m]]*.

[n[,m]]{*command...*}
Execute each enclosed *command* on the given lines.

:*label* Define *label* for use in branch or test command.

*[n]***a**\ Append new text after given line. New text is terminated by any line not ending in '\'.

b *[label]*
Branch to *label*, which must be defined in a ':' command. If *label* is omitted, branch to end of command script.

*[n[,m]]***c**\
Change specified lines to new text and proceed with next input line. New text is terminated by any line not ending in '\'.

*[n[,m]]***d** Delete specified lines and proceed with next input line.

*[n[,m]]***D** Delete first line in work area and proceed with next input line.

*[n[,m]]***g** Copy secondary work area to work area, destroying previous contents.

*[n[,m]]***G** Append secondary work area to work area.

*[n[,m]]***h** Copy work area to secondary work area, destroying previous contents.

*[n[,m]]***H** Append work area to secondary work area.

*[n]***i**\ Insert new text before given line. New text is terminated by any line not ending in '\'.

*[n[,m]]***l** Print selected lines, interpreting non-graphic characters.

*[n[,m]]***n** Print the work area and replace it with the next input line.

*[n[,m]]***N** Append next input line preceded by a newline to work area.

*[n[,m]]***p** Print work area.

*[n[,m]]***P** Print first line of work area.

*[n]***q** Quit without reading any more input.

*[n]***r** *file* Copy *file* to output.

*[n[,m]]***s***[k]/pattern1/pattern2/*[**g**][**p**][**w** *file]*
Search for *pattern1* and substitute *pattern2* for *k*th occurrence (default, first). If optional **g** is given, substitute all occurrences. If optional **p** is given, print the resulting line. If optional **w** is given, append the resulting line to *file*. Patterns are described in detail below.

*[n[,m]]***t***[label]*
Test if substitutions have been made. If so, branch to *label*, which must be defined in a ':' command. If *label* is omitted, branch to end of command script.

*[n[,m]]***w** *file*
Append lines to *file*.

[n[,m]] **x**
Exchange the work area and the secondary work area.

LEXICON

*[n[,m]]***y***/chars/replacements/*
Translate characters in *chars* to the corresponding characters in *replacements*.

Patterns

Substitution commands and search specifications may include *patterns*, also called *regular expressions*. Pattern specifications are identical to those of **ed**, except that the special characters '**n**' match a newline character in the input.

A non-special character in a pattern matches itself. Special characters include the following:

^ Match beginning of line, unless it appears immediately after '[' (see below).

$ Match end of line.

n Match the newline character.

. Match any character except newline.

* Match zero or more repetitions of preceding character.

[*chars*] Match any one of the enclosed *chars*. Ranges of letters or digits may be indicated using '-'.

[^*chars*]
Match any character *except* one of the enclosed *chars*. Ranges of letters or digits may be indicated using '-'.

c Disregard special meaning of character *c*.

\(*pattern*\)
Delimit substring *pattern*; for use with *d*, described below.

In addition, the replacement part *pattern2* of the substitute command may also use the following:

& Insert characters matched by *pattern1*.

d Insert substring delimited by *d*th occurrence of delimiters '\(' and '\)', where *d* is a digit.

Options

sed recognizes the following options:

-e Next argument gives commands.

-f Next argument gives file name of command script.

-n Output lines only when explicit **p** or **P** commands are given.

See Also

commands, ed, elvis, ex, me, vi

seg.h — Header File

Definitions used with segmentation
#include <seg.h>

seg.h defines structures and constants used by routines that handle memory segmentation.

See Also
header files

sem — Device Driver
Semaphore device driver

/dev/sem is an interface to the semaphore device driver. It is assigned major device 23 (minor device 0) and can be accessed as a character-special device.

All semaphore operations are performed through the COHERENT system call **ioctl**. The operations **semctl**, **semget**, and **semop** are performed with an integer parameter array. The first element of the array is reserved for the return value (default -1). Subsequent elements represent arguments. The call to **ioctl** passes **SEMCTL**, **SEMGET**, or **SEMOP** as the second argument, and the parameter array as the third argument. The first argument is an open file descriptor to **/dev/sem**.

Access
Prior to accessing the devices, a entry must be created in **/dev**, as follows:

```
/etc/mknod /dev/sem c 23 0
chmod 666 /dev/sem
```

Files
/usr/include/sys/ipc.h
/usr/include/sys/sem.h
/dev/sem — Device
/drv/sem — Loadable device driver

See Also
device drivers, drvld, semctl(), semget(), semop()

Notes
Allocation of too many semaphore ids (**NSEMID**) or semaphores per identifier (**NSEM**) can exhaust kernel data space, which will stop the system in its tracks.

Private semaphore sets are not supported. Semaphore ids must be removed manually when no longer required. To remove all semaphore identifiers, use the following code:

```
#include <sys/sem.h>
#define NSEMID 16

semget( 0, 0 );     /* must do first */

for ( id=0; id < NSEMID; ++id )
        semctl( id, 0, IPC_RMID, 0 );
```

To load the driver **sem** into memory, use the command **drvld**.

sem.h — Header File
Definitions used by semaphore facility
#include <sys/sem.h>

sem.h defines constants and structures used by the COHERENT semaphore facility.

See Also

header files

semctl() — COHERENT System Call

Control semaphore operations

```
#include <sys/sem.h>
semctl(semid, semnum, cmd, arg)
int semid, cmd, semnum;
union semun {
        int val;
        struct semid_ds *buf;
        unsigned short array[];
} arg;
```

semctl controls a variety of semaphore operations. *cmd* sets the operation to be performed; the following *cmd*s are executed with respect to the semaphore specified by *semid* and *semnum*:

GETVAL Return the value of **semval** (READ).

SETVAL Set the value of **semval** to *arg.val* (ALTER).

GETPID Return the value of **sempid** (READ).

GETNCNT Return the value of **semncnt** (READ).

GETZCNT Return the value of **semzcnt** (READ).

The following *cmd*s return and set, respectively, every semval in the set of semaphores.

GETALL Place semvals into array pointed to by *arg.array* (READ).

SETALL Set *semvals* according to the array pointed to by *arg.array* (ALTER).

The following *cmd*s are also available:

IPC_STAT Place the current value of each member of the data structure associated with *semid* into the structure pointed to by *arg.buf* (READ).

IPC_SET Set the value of the following members of the data structure associated with *semid* to the corresponding value found in the structure pointed to by *arg.buf*:

```
sem_perm.uid
sem_perm.gid
sem_perm.mode /* only low 9 bits */
```

This command can only be executed by a process that has an effective user identifier equal to either that of superuser or to the value of **sem_perm.uid** in the data structure associated with *semid*.

IPC_RMID Remove the system identifier specified by *semid* from the system and destroy the set of semaphores and data structure associated with it. This *cmd* can only be executed by a process that has an effective user identifier equal to either that of super user or to the value of **sem_perm.uid** in the data structure associated with *semid*.

semctl will fail if one or more of the following are true:

- *semid* is not a valid semaphore identifier [EINVAL].
- *semnum* is less than zero or greater than **sem_nsems** [EINVAL].
- *cmd* is not a valid command [EINVAL].
- Operation permission is denied to the calling process. [EACCES]
- *cmd* is **SETVAL** or **SETALL** and the value to which semval is to be set is greater than the system imposed maximum [ERANGE].
- *cmd* is equal to **IPC_RMID** or **IPC_SET** and the effective user identifier of the calling process is not equal to that of superuser and it is not equal to the value of **sem_perm.uid** in the data structure associated with *semid* [EPERM].
- *arg.buf* points to an illegal address [EFAULT].

Return Value

Upon successful completion, the value returned depends on *cmd* as follows:

GETVAL	The value of **semval**.
GETPID	The value of **sempid**.
GETNCNT	The value of **semncnt**.
GETZCNT	The value of **semzcnt**.
All others	Zero

Otherwise, **semctl** returns -1 and sets **errno** to an appropriate value.

Files

/usr/include/sys/ipc.h
/usr/include/sys/sem.h
/dev/sem
/drv/sem

See Also

COHERENT system calls, sem, semget(), semop()

Notes

To improve portability, the COHERENT system implements the semaphore functions as a device driver rather than as an actual system call.

semget() — COHERENT System Call

Get a set of semaphores
#include <sys/sem.h>
semget(*key*, *nsems*, *semflg*)
key_t *key*; **int** *nsems*, *semflg*;

semget returns the semaphore identifier associated with *key*. It creates a semaphore identifier and associated data structure and set that contains *nsems* semaphores for *key* should one of the following be true:

- *key* equals **IPC_PRIVATE**.
- *key* does not have a semaphore identifier associated with it, and (*semflg* & **IPC_CREAT**) is true.

When **semget** creates a data structure for a new semaphore identifier, it initializes the structure as follows:

- It sets the fields **sem_perm.cuid**, **sem_perm.uid**, **sem_perm.cgid**, and **sem_perm.gid** equal to the effective user identifier, the calling process's identifier, and the effective group identifier, respectively.
- It sets the low-order nine bits of **sem_perm.mode** equal to the low-order nine bits of *semflg*. These nine bits define access permissions: the top three bits specify the owner's access permissions (read, write, execute), the middle three bits the owning group's access permissions, and the low three bits access permissions for others.
- **sem_nsems** is set equal to the value of *nsems*.
- **sem_otime** is set to zero and **sem_ctime** to the current time.

semget fails if any of the following are true:

- *nsems* is either less than or equal to zero, or greater than the system imposed limit. It sets **errno** to **EINVAL**.
- A semaphore identifier exists for *key* but operation permission as specified by the low-order nine bits of *semflg* would not be granted (**EACCES**).
- A semaphore identifier exists for *key* but the number of semaphores in the set associated with it is less than *nsems* and *nsems* is not equal to zero (**EINVAL**).
- A semaphore identifier does not exist for *key* and (*semflg* & **IPC_CREAT**) is false (**ENOENT**).
- The number of semaphore identifiers allowed system-wide would be exceeded (**ENOSPC**).
- The number of semaphores allowed system-wide would be exceeded (**ENOSPC**).
- A semaphore identifier exists for *key* but ((*semflg* & **IPC_CREAT**) **&&** (*semflg* & **IPC_EXCL**)) is true (**EEXIST**).

Return Value

Upon successful completion, **semget** returns a non-negative integer, namely a semaphore identifier. Otherwise, it returns -1 and sets **errno** to an appropriate value.

Files

/usr/include/sys/ipc.h
/usr/include/sys/sem.h
/dev/sem
/drv/sem

See Also

COHERENT system calls, sem, semctl(), semop()

Notes

To improve portability, the COHERENT system implements the semaphore functions as a device driver rather than as an actual system call.

semop() — COHERENT System Call

Perform semaphore operations

#include <sys/sem.h>
semop(*semid, sops, nsops***)**
int *semid, nsops*; **struct sembuf(***sops***)[];**

semop can atomically perform a number of operations on the set of semaphores associated with the semaphore identifier *semid*. *sops* pointer to the array of semaphore-operation structures. *nsops* is the number of such structures in the array. Each structure includes the following members:

```
short   sem_num;    /* semaphore number */
short   sem_op;     /* semaphore operation */
short   sem_flg;    /* operation flags */
```

Each semaphore operation specified by *sem_op* is performed on the semaphore specified by *semid* and *sem_num*.

sem_op specifies one of three semaphore operations, as follows:

- If *sem_op* is negative, one of the following occurs:
  1. If **semval** is greater than or equal to the absolute value of *sem_op*, the absolute value of *sem_op* is subtracted from **semval**.
  2. If **semval** is less than the absolute value of *sem_op* and (*sem_flg* & **IPC_NOWAIT**) is true, **semop** sets **errno** to **EGAIN** and returns -1.
  3. If **semval** is less than the absolute value of *sem_op* and (*sem_flg* & **IPC_NOWAIT**) is false, **semop** increments the **semncnt** associated with the specified semaphore and suspend execution of the calling process until one of the following occurs:
     a. **semval** becomes greater than or equal to the absolute value of *sem_op*. When this occurs, the value of **semncnt** associated with the specified semaphore is decremented, and the absolute value of *sem_op* is subtracted from **semval**.
     b. The *semid* for which the calling process is awaiting action is removed from the system.
     c. The calling process receives a signal. When this occurs, the value of **semncnt** associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in *signal*.
- If *sem_op* is positive, the value of *sem_op* is added to **semval**.
- If *sem_op* is zero, one of the following occurs:
  1. If **semval** is zero, **semop** returns immediately.
  2. If **semval** does not equal zero and (*sem_flg* & **IPC_NOWAIT**) is true, **semop** immediately returns -1, with **errno** set to **EGAIN**.
  3. If **semval** is not equal to zero and (*sem_flg* & **IPC_NOWAIT**) is false, **semop** increments the **semzcnt** associated with the specified semaphore and suspends execution of the calling process until one of the following occurs:
     a. **semval** becomes zero, at which time the value of **semzcnt** associated with the specified semaphore is decremented.
     b. The *semid* for which the calling process is awaiting action is removed from the system.
     c. The calling process receives a signal. When this occurs, the value of **semzcnt** associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in *signal*.

semop fails if one or more of the following are true for any of the semaphore operations specified by *sops*:

- *semid* is not a valid semaphore identifier. **semop** sets **errno** to **EINVAL**
- *sem_num* is less than zero or greater than or equal to the number of semaphores in the set associated with *semid* (**EFBIG**).
- *nsops* is greater than the system imposed maximum (**E2BIG**).
- Operation permission is denied to the calling process (**EACCES**).
- The operation would result in suspension of the calling process but (*sem_flg* & **IPC_NOWAIT**) is true (**EAGAIN**).
- An operation would cause a **semval** to overflow the system imposed limit (**ERANGE**).
- *sops* points to an illegal address (**EFAULT**).

Upon successful completion, the value of **sempid** for each semaphore specified in the array pointed to by *sops* is set equal to the process identifier of the calling process.

Return Value

If **semop** returns due to the receipt of a signal, it returns -1 to the calling process and sets **errno** to **EINTR**. If it returns due to the removal of a *semid* from the system, it returns -1 and sets **errno** to **EDOM**.

Upon successful completion, **semop** returns the value of **semval** at the time of the call for the last operation in the array pointed to by *sops*. Otherwise, it returns -1 and sets **errno** to an appropriate value.

Files

/usr/include/sys/ipc.h
/usr/include/sys/sem.h
/dev/sem
/drv/sem

See Also

COHERENT system calls, sem, semctl(), semget()

Notes

The flag **SEM_UNDO** is not supported. This flag would allow semaphore operations to be undone upon the termination of the process which performed the operations.

To improve portability, the COHERENT system implements semaphore operations as a device driver rather than as an actual system call.

set — Command

Set shell option flags and positional parameters
set [**-ceiknstuvx** [*name* ...]] (Bourne shell)
set [[+-]**aefhkmnuvx**] [[+-]**o** *name*] (Korn shell)

set changes the options of the current shell and optionally sets the values of positional parameters. This command is used implemented by both the Bourne and Korn shells; however, its syntax and options vary from one shell to the other.

Bourne Shell

The shell variable '$-' contains the currently set shell flags. If the optional *name* list is given, **set** assigns the positional parameters **$1**, **$2** ... to the given shell variables.

set recognizes the following options:

-c *string*
: Read shell commands from *string*.

-e Exit on any error (command not found or command returning nonzero status) if the shell is not interactive.

-i The shell is interactive, even if the terminal is not attached to it; print prompt strings. For a shell reading a script, ignore signals **SIGTERM** and **SIGINT**.

-k Place all keyword arguments into the environment. Normally, the shell places only assignments to variables preceding the command into the environment.

-n Read commands but do not execute them.

-s Read commands from the standard input and write shell output to the standard error.

-t Read and execute one command rather than the entire file.

-u If the actual value of a shell variable is blank, report an error rather than substituting the null string.

-v Print each line as it is read.

-x Print each command and its arguments as it is executed.

- Cancel the **-x -v** options.

The shell executes **set** directly.

Korn Shell

set recognizes the following options. Preceding an option with '-' turns on the option; preceding it with '+' turns it off.

-a **allexport**: Automatically export all new variables.

-e **errexit**: Exit from the shell when non-zero status is received.

-f **noglob**: Do not expand file names. This globally turns off the special meaning of characters '*' and '?'.

-h **trackall**: Automatically add all commands to the shell's hash table.

-k **keyword**: Recognize variable assignments anywhere in a command.

-m **monitor**: Enable job control. See the Lexicon article on **ksh** for details on job control and how to use it.

-n **noexec**: Compile an input command, but do not execute it.

-o *option*
: Set *option*. **set** recognizes the following *options*:

allexport Same as **-a** option, above.

emacs Turn on MicroEMACS-style editing of command lines.

errexit Same as **-e** option, above.

ignoreeof Tell the shell not to exit when reading EOF: must use **exit** command to exit from the shell.

keyword Same as **-k** option, above.

monitor Same as **-m** option, above.

noexec Same as **-n** option, above.

noglob Same as **-f** option, above.

trackall Same as **-h** option, above.

nounset Same as **-u** option, below.

verbose Same as **-v** option, below.

xtrace Same as **-x** option, below.

-u **nounset**: Treat dollar-sign expansion of an unset variable as an error.

-v **verbose**: When compiling a command, echo its compiled (i.e., expanded) version on the standard output before executing it.

-x **xtrace**: Echo simple commands while executing.

The shells execute **set** directly.

See Also

commands, ksh, sh

setbuf() — STDIO (libc)

Set alternative stream buffers
#include <stdio.h>
void setbuf(*fp*, *buffer*) **FILE ***fp*; **char ***buffer*;

The standard I/O library STDIO automatically buffers all data read and written in streams, with the exception of streams to terminal devices. STDIO normally uses **malloc** to allocate the buffer, which is a **char** array **BUFSIZ** characters long; **BUFSIZ** is defined in the header file **stdio.h**.

setbuf's arguments are the file stream *fp* and the *buffer* to be associated with the stream. The call should be issued after the stream has been opened, but before any input or output request has been issued. If *buffer* is NULL, the stream will be unbuffered. If *buffer* is not NULL, the area of memory it points to must contain at least **BUFSIZ** bytes.

setbuf returns nothing.

See Also

fopen(), malloc(), STDIO

setgid() — COHERENT System Call

Set group id and user id
int setgid(*id***) int** *id*;

setgid sets the group id. This calls can be used to set group id privileges. (For more information on group id, see **exec**.)

The call is allowed if the real id of the calling process matches *id* or is the superuser.

See Also

COHERENT system calls, exec, getuid(), login, setuid()

Diagnostics

setgid returns zero on success, or -1 on failure.

setgrent() — General Function (libc)

Rewind group file
#include <grp.h>
struct group *setgrent();

setgrent rewinds the file **/etc/group**. It returns NULL if an error occurs.

Files

/etc/group
<grp.h>

See Also

general functions, group

setjmp() — General Function (libc)

Perform non-local goto
#include <setjmp.h>
int setjmp(*env***) jmp_buf** *env***;**

The function call is the only mechanism that C provides to transfer control between functions. This mechanism, however, is inadequate for some purposes, such as handling unexpected errors or interrupts at lower levels of a program. To answer this need, **setjmp** helps to provide a non-local *goto* facility. **setjmp** saves a stack context in *env*, and returns value zero. The stack context can be restored with the function **longjmp**. The type declaration for **jmp_buf** is in the header file **setjmp.h**. The context saved includes the program counter, stack pointer, and stack frame.

Example

The following gives a simple example of **setjmp** and **longjmp**.

```
#include <setjmp.h>

jmp_buf env;    /* place for setjmp to store its environment */

main()
{
     int rc;

     if(rc = setjmp(env)) { /* we come here on return */
          printf("First char was %c\n", rc);
          exit(0);
     }
     subfun();  /* this never returns */
}

subfun()
{
     char buf[80];
```

```
        do {
            printf("Enter some data\n");
            gets(buf); /* get data */
        } while(!buf[0]);     /* retry on null line */

        longjmp(env, buf[0]);   /* buf[0] must be non zero */
}
```

See Also

general functions, getenv(), longjmp()

Notes

Programmers should note that many user-level routines cannot be interrupted and reentered safely. For that reason, improper use of **setjmp** and **longjmp** can create mysterious and irreproducible bugs. The use of **longjmp** to exit interrupt exception or signal handlers is particularly hazardous.

setjmp.h — Header File

Define setjmp() and longjmp()
#include <setjmp.h>

setjmp.h defines the structure **jmp_buf** for a **setjmp** environment.

See Also

header file, longjmp, setjmp

setpwent() — General Function (libc)

Rewind password file
#include <pwd.h>
setpwent()

The COHERENT system has five routines that search the file **/etc/passwd**, which contains information about every user of the system. The returned structure **passwd** is defined in the header file **pwd.h**, as follows:

```
struct passwd {
        char    *pw_name;     /* login user name */
        char    *pw_passwd;   /* login password */
        int     pw_uid;       /* login user id */
        int     pw_gid;       /* login group id */
        int     pw_quota;     /* file quota (unused) */
        char    *pw_comment;  /* comments (unused) */
        char    *pw_gecos;    /* (unused) */
        char    *pw_dir;      /* working directory */
        char    *pw_shell;    /* initial program */
};
```

For detailed descriptions of the above fields, see the entry for **passwd**.

setpwent rewinds the password file, which allows searches to begin from the beginning of the file.

Files

/etc/passwd
<pwd.h>

See Also

endpwent(), general functions, getpwent(), getpwnam(), getpwuid(), pwd.h

Diagnostics

setpwent returns NULL for any error.

settz() — Time Function

Set local time zone
#include <time.h>
#include <sys/types.h>
void settz()
extern long timezone; char *tzname[2][16];

settz is one of the suite of COHERENT functions that control and display the system's time. It searches for the environmental parameter **TIMEZONE**, which gives information on the local time zone. For more information on **TIMEZONE**, see its Lexicon entry.

If **TIMEZONE** is set, **settz** initializes the external variables **timezone** and **tzname**. **timezone** contains the number of seconds to be subtracted from GMT to obtain local standard time. **tzname[0]** and **tzname[1]** are character arrays that hold, respectively, the names of the local standard time zone and the local daylight saving time zone. If **TIMEZONE** is not set, **timezone** defaults to 0, **tzname[0]** to **GMT**, and **tzname[1]** to the empty string.

See Also

date, ftime(), time, TIMEZONE

setuid() — COHERENT System Call

Set user id
int setuid(*id*) int *id*;

setuid sets the real user id and the user id of the calling process to *id*. (For more information on the user id, see **exec**).

The call is allowed if the real id of the calling process matches *id* or is the superuser.

See Also

COHERENT system calls, exec, getuid(), login, setgid()

Diagnostics

setuid returns zero on success, or -1 on failure.

sgtty — Device Driver

General terminal interface

Under the COHERENT system, all asynchronous ports use the same interface, no matter what hardware is involved. The remainder of this section discusses the common features of this interface.

When a terminal file is opened, it normally causes the process to wait until a connection is established. In practice, users' programs seldom open these files; they are opened by the program *getty* and become a user's standard input, output, and error files. The very first terminal file opened by the process group leader of a terminal file not already associated

with a process group becomes the *controlling terminal* for that process group. The controlling terminal plays a special role in handling **quit** and **interrupt** signals, as discussed below. The controlling terminal is inherited by a child process during a call to **fork**. A process can break this association by changing its process group using **setpgrp**.

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters can be typed at any time, even while output is occurring, and are only lost when the system's input buffers become completely full, which is rare, or when the user has accumulated the maximum allowed number of input characters that have not yet been read by some program. Currently, this limit is 256 characters. When the input limit is reached, the sytems throws away all the saved characters without notice.

Normally, terminal input is processed in units of lines. A line is delimited by a newline character (ASCII LF) or an end-of-file character (ASCII EOT). Unless otherwise directed, a program attempting to read will be suspended until an entire line has been typed. Also, no matter how many characters are requested in the read call, at most one line will be returned. It is not, however, necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information.

During input, the system normally processes **erase** and **kill** characters. By default, the backspace character erases the last character typed, except that it will not erase beyond the beginning of the line. By default, the **<ctrl-U>** kills (deletes) the entire input line, and optionally outputs a newline character. Both these characters operate on a keystroke-by-keystroke basis, independently of any backspacing or tabbing which may have been done. Both the erase and kill characters may be entered literally by preceding them with the escape character (\). In this case, the escape character is not read. You may change the erase and kill characters via command **stty**.

Certain characters have special functions on input. These functions and their default character values are summarized as follows:

INTR (**<ctrl-C>** or ASCII ETX) generates an *interrupt* signal that is sent to all processes associated with the controlling terminal. Normally, each such process is forced to terminate, but arrangements may be made either to ignore the signal or to receive a trap to an agreed-upon location; see the Lexicon entry for **signal**.

QUIT (Control-\ or ASCII ES) generates a *quit* signal. Its treatment is identical to that of the interrupt signal except that, unless a receiving process has made other arrangements, it will not only be terminated but a core image file (called **core**) will be created in the current working directory.

ERASE (**<backspace>** or ASCII BS) erases the preceding character. It will not erase beyond the start of a line, as delimited by a newline or **EOF** character.

KILL (**<ctrl-U>** or ASCII NAK) deletes the entire line, as delimited by a newline or **EOF** character.

EOF (**<ctrl-D>** or ASCII EOT) generates an end-of-file character from a terminal. When received, all the characters waiting to be read are immediately passed to the program without waiting for a newline, and the **EOF** is discarded. Thus, if no characters are waiting, which is to say the **EOF** occurred at the beginning of a line. zero characters will be passed back, which is the standard end-of-file indication.

NL (ASCII LF) is the normal line delimiter. It cannot be changed or escaped.

STOP (**<ctrl-S>** or ASCII DC3) can be used to suspend output. It is useful with CRT terminals to prevent output from disappearing before it can be read. While output is suspended, STOP characters are ignored and not read.

START (**<ctrl-Q>** or ASCII DC1) resumes output that has been suspended by a STOP character. While output is not suspended, START characters are ignored and not read. The start/stop characters can be changed via command **stty**, or via special **ioctl()** calls described below.

The character values for INTR, QUIT, ERASE, **EOF**, and KILL may be changed to suit individual tastes. The ERASE, KILL, and **EOF** character may be escaped by a preceding \ character, in which case the system ignores its special meaning. See the Lexicon article on **stty** for information on how to change these settings dynamically.

When using a "modem control" serial line, loss of carrier from the data-set (modem) causes a *hangup* signal to be sent to all processes that have this terminal as the controlling terminal. Unless other arrangements have been made, this signal causes the process to terminate. If the hangup signal is ignored, any subsequent read returns with an end-of-file indication. Thus programs that read a terminal and test for end-of-file can terminate appropriately when hung up on.

When one or more characters are written, they are transmitted to the terminal as soon as previously written characters have finished typing. Input characters are echoed by putting them into the output queue as they arrive. If a process produces characters more rapidly than they can be printed, it will be suspended when its output queue exceeds some limit, known as the "high water mark". When the queue has "drained" down to some threshold, the program resumes.

The header file **<sgtty.h>** declares structures and manifest constants to control the **sgtty** interface. Of interest to users are the constants that define baud rates for terminal ports; these are as follows:

B50	50 baud
B75	75 baud
B110	110 baud
B134	134 baud
B150	150 baud
B200	200 baud
B300	300 baud
B600	600 baud
B1200	1200 baud
B1800	1800 baud
B2000	2000 baud
B2400	2400 baud
B3600	3600 baud
B4800	4800 baud
B7200	7200 baud
B9600	9600 baud
B19200	19,200 baud

Terminal ioctl() Functions

Header file **<sgtty.h>** defines the following data structures used by the various device drivers to convey terminal specific information. These structures are used in conjunction with special terminal or device driver symbolic constants as part of **ioctl()** requests.

The **sgttyb** structure contains information related to line discipline, such as serial line speed, if appropriate, the "erase" and "kill" characters, and a series of flags which set the

mode of the line.

```
/*
 * Structure for TIOCSETP/TIOCGETP
 */
struct sgttyb {
        char   sg_ispeed;    /* Input speed */
        char   sg_ospeed;    /* Output speed */
        char   sg_erase;     /* Character erase */
        char   sg_kill;      /* Line kill character */
        int    sg_flags;     /* Flags */
};
```

The following symbolic constants are used to access bit positions of member **sg_flags** in data structure **sgttyb**:

CBREAK Each input character causes wakeup (i.e., forces a return from a **read()** system call).

CRMOD Map the carriage return characters '\r' to the newline character '\n'.

CRT Use CRT-style character erase.

ECHO Echo input characters.

EVENP Select even parity. If used in conjunction with **ODDP**, allow either parity.

LCASE Lowercase mapping on input.

ODDP Select odd parity. If used in conjunction with **EVENP**, allow either parity.

RAW Raw mode. Same as **RAWIN** plus **RAWOUT**.

RAWIN Input is treated as 8-bit characters and not interpreted.

RAWOUT Output is treated as 8-bit characters and not interpreted.

TANDEM Use X-ON/X-OFF flow control protocol to remote device.

XTABS Expand tabs to spaces.

Data structure **tchars** specifies additional special terminal characters such as the "interrupt" and "quit" characters, the "start" and "stop" characters used for flow control, and the "end-of-file" character.

```
/*
 * Structure for TIOCSETC/TIOCGETC
 */
struct tchars {
        char   t_intrc;      /* Interrupt */
        char   t_quitc;      /* Quit */
        char   t_startc;     /* Start output */
        char   t_stopc;      /* Stop output */
        char   t_eofc;       /* End of file */
        char   t_brkc;       /* Input delimiter */
};
```

The following symbolic constants are used to access various device functions via **ioctl()** calls, as defined in header file **<sgtty.h>**. Note that not all functions are appropriate for all classes of devices.

TIOCCBRK Clear a BREAK condition on a serial line (i.e., "mark" the line). This request cancels a previously issued **TIOCSBRK** request.

TIOCCDTR Clear modem control signal Data Terminal Ready (DTR) on a serial line.

TIOCCHPCL Do not force a hangup on "last close" on a modem line. The normal mode of operation for serial lines is to drop modem signal Data Terminal Ready (DTR) when the last **close()** operation is performed, thus requesting the attached modem to drop the connection.

TIOCCRTS Clear the Request To Send (RTS) signal on a serial line. Modem control signal RTS is often used for hardware flow control.

TIOCEXCL Set device access as exclusive use. This request requires the process to have **root** privileges.

TIOCFLUSH Flush the input queue, discarding any pending input characters, and wait for the output queue to "drain".

TIOCGETC Get current values of the special terminal characters, as defined by data structure **tchars**.

TIOCGETF Get current console keyboard function key bindings. This request is specific to the **nkb** console keyboard device driver. See Lexicon article **nkb** for further details.

TIOCGETKBT Get current console keyboard key mapping table. This request is specific to the **nkb** console keyboard device driver. See Lexicon article **nkb** for further details.

TIOCGETP Get current terminal line settings, as defined by data structure **sgttyb**.

TIOCGETTF Get current value of the terminal flags, as defined by field **t_flags** in the TTY structure.

TIOCHPCL Set hangup on "last close". See **TIOCCHPCL** for further details.

TIOCRMSR Get the current value of the Modem Status Register (MSR) for the specified serial line. This request is device driver specific and is currently supported only in the **al** device driver. Symbolic constants **MSRCTS**, **MSRDSR**, **MSRRI**, and **MSRRLSD** correspond to the Clear To Send, Data Set Ready, Ring Indicator and Receive Line Status Detect

(i.e. Carrier Detect) signals, respectively, in the MSR.

TIOCNXCL Set this device or port as non-exclusive use. See **TIOCEXCL** for further details.

TIOCQUERY Query the number of characters currently waiting in the input queue.

TIOCSBRK Assert BREAK (i.e., "space the line") on the given serial port. This is often used during login to signal a remote system to "hunt" to the next baud rate in a sequence. See **TIOCCBRK** for further details.

TIOCSDTR Assert modem control signal Data Terminal Ready (DTR) on a serial line.

TIOCSETC Wait for output to "drain", then set the terminal control characters for this device, as specified by data structure **tchars**.

TIOCSETF Set console keyboard function key mapping. This request is specific to the **nkb** console keyboard device driver. See Lexicon article **nkb** for further details.

TIOCSETKBT Set console keyboard key mapping table. This request is specific to the **nkb** console keyboard device driver. See Lexicon article **nkb** for further details.

TIOCSETN Set terminal line settings, as defined by data structure **sgttyb**. Do not flush the input queue prior to using the new settings.

TIOCSETP Same as request **TIOCSETN**, but also flush the input queue.

TIOCSRTS Assert the Request To Send (RTS) signal on a serial line. Modem control signal RTS is often used for hardware flow control.

Examples

The following code fragment gets the current terminal settings and turns off echo.

```
#include <sgtty.h>
static struct sgttyb new, orig;
. . .
/*
 * Get the existing terminal parameters for the terminal
 * device associated with file descriptor 0 (stdin),
 * turn off echo, turn on CBREAK (break on every input character)
 * and set the new parameters.
 */
(void) ioctl(0, TIOCGETP, &orig);
new = orig;
new.sg_flags &= ~ECHO;          /* Turn off echo */
new.sg_flags |= CBREAK;         /* Turn on CBREAK mode */
(void) ioctl(0, TIOCSETN, &new);
```

The following line uses the previously saved terminal mode to return the terminal mode to its prior state:

```
(void) ioctl(0, TIOCSETN, &orig);
```

See Also

device drivers, gtty(), ioctl(), sgtty.h, stty(), stty, terminal

sgtty.h — Header File

Definitions used to control terminal I/O
#include <sgtty.h>

sgtty.h defines structures, constants, and macros used by routines that control terminal I/O.

See Also

header files, sgtty

Notes

Note that for historical reasons, the set of baud rates defined in header file **termio.h** differs from the ones defined in **sgtty.h**. If you **#include <termio.h>** before doing terminal ioctl's, you will get the wrong baud rates.

sh — Command

The Bourne shell
sh [-ceiknstuvx] *token* ...

The COHERENT system offers two command interpreters: **ksh**, the Korn shell; and **sh**, the Bourne shell. **sh** is the default COHERENT command interpreter. The tutorial included in this manual describes the Bourne shell in detail.

As you will see from the following description, a shell is both a command interpreter and a programming language in its own right. Taking some time to learn the rudiments of your shell's programming language will pay great benefits in taking command of your COHERENT system.

Commands

A *command* consists of one or more *tokens*. A *token* is a string of text characters (i.e., one or more alphabetic characters, punctuation marks, and numerals) delineated by spaces, tabs, or newlines.

A *simple command* consists of the command's name, followed by zero or more tokens that represent represent arguments to the command, names of files, or shell operators. A *complex command* will use shell constructs to execute one or more commands conditionally. In effect, a complex command is a mini-program that is written inthe shell's programming language and interpreted by **sh**.

Shell Operators

The shell includes a number of operators that form pipes, redirect input and output to commands, and let you define conditions under which commands are executed.

command | *command*

The *pipe* operator: let the output of one command serve as the input to a second. You can combine commands with '|' to form *pipelines*. A pipeline passes the standard output of the first (leftmost) command to the standard input of the second command. For example, in the pipeline

```
sort customers | uniq | more
```

sh invokes **sort**B to sort the contents of file **customers**. It pipes the output of **sort** to the command **uniq**, which outputs one unique copy of the text that is input into it. **sh** then pipes the output of **uniq** to the command **more**, which displays it on your terminal one screenful at a time. Note that under COHERENT, unlike MS-

DOS, pipes are executed concurrently: that is, **sort** does not have to finish its work before **uniq** and **more** can begin to receive input and get to to work. passes the standard output of

command **;** *command*
Execute commands on a command line sequentially. The command to the left of the ';' executes to completion; then the command to the right of it executes. For example, in the command line

```
a | b ; c | d
```

first executes the pipeline **a | b** then, when **a** and **b** are finished, executes the pipeline **c | d**.

command **&** *command*
Execute a command in the background. This operator must follow the command, not precede it. It prints the process identifier of the command on the standard output, so you can use the **kill** command to kill that process should something go wrong. This operator lets you execute more than one command simultaneously. For example, the command

```
fdformat -v /dev/fha0 &
```

formats a high-density, 5.25-inch floppy disk in drive 0 (that is, drive A); but while the disk is being formatted, **sh** returns the command line prompt so you can immediately enter another command and begin to work. If you did not use the '&' in this command, you would have to wait until formatting was finished before you could enter another command.

command **&&** *command*
Execute a command upon success. **sh** executes the command that follows the token '&&' only if the commands that precedes it returns a zero exit status, which signifies success. For example, the command

```
cd /etc
fdformat -v /dev/fha0 && badscan -o proto /dev/fha0 2400
```

formats a floppy disk, as described above. If the format was successful, it then invokes the command **badscan** to scan the disk for bad blocks; if it was not successful, however, it does nothing.

command **||** *command*
Execute a command upon failure. This is identical to operator '&&', except that the second command is executed if the first returns a non-zero status, which signifies failure. For example, the command

```
/etc/fdformat -v /dev/fha0 || echo "Format failed!"
```

formats a floppy disk. If formatting failed, it echoes the message **Format failed!** on your terminal; however, if formatting succeeds, it does nothing.

Note that the tokens newline, ';' and '&' bind less tightly than '&&' and '||'. **sh** parses command lines from left to right if separators bind equally.

>*file* Redirect output. The *standard input*, *standard output*, and *standard error* streams are normally connected to the terminal. A pipeline attaches the output of one command to the input of another command. In addition, **sh** includes a set of operators that redirect input and output into files rather than other commands.

The operator '>' redirects output into a file. For example, the command

```
sort customers >customers.sort
```

sorts file **customers** and writes the sorted output into file **customers.sort**. It creates **customers.sort** if it does not exist, and destroys its previous contents if it does exist.

>>file Redirect output into a file, and append. If the file does not exist, this operator creates it; however, if the file already exists, this operator appends the output to that file's contents rather than destroying those contents. For example, the command

```
sort customers.now | uniq >>customers.all
```

sorts file **customers.now**, pipes its output to command **uniq**, which throws away duplicate lines of input, and appends the results to file **customers.all**.

<file Redirect input. Here, **sh** reads the contents of a file and processes them as if you had typed them from your keyboard. For example, the command

```
ed textfile <edit.script
```

invokes the line-editor **ed** to edit **textfile**; however, instead of reading editing commands from your keyboard, the shell passes **ed** the contents of **edit.script**. This command would let you prepare an editing script that you could execute repeatedly upon files rather than having to type the same commands over and over.

<< token

Prepare a "here document". This operator tells **sh** to accept standard input from the shell input until it reads the next line that contains only *token*. For example, the command

```
cat >FOO <<\!
     Here is some text.
!
```

redirects all text between '<<\!' and '!' to the **cat** command. The '>' in turn redirects the output of **cat** into file **FOO**. **sh** performs parameter substitution on the here document unless the leading *token* is quoted; parameter substitution and quoting are described below.

command **2>** *file*

Redirect the standard error stream into a file. For example, the command

```
nroff -ms textfile >textfile.p 2>textfile.err
```

invokes the command **nroff** to format the contents of **textfile**. It redirects the output of **nroff** (i.e., the standard output) into **textfile.p**; it also redirects any error messages that **nroff** may generate into file **textfile.err**.

Note in passing that a command may use up to 20 streams. By default, stream 0 is the standard input; stream 1 is the standard output; and stream 2 is the standard error. **sh** lets you redirect any of these streams individually into files, or combine streams into each other.

<&n **sh** can redirect the standard input and output to duplicate other file descriptors. (See the Lexicon article **file descriptor** for details on what these are.) This operator duplicates the standard input from file descriptor *n*.

>&*n* Duplicate the standard output from file descriptor *n*. For example,

```
2>&1
```

redirects file descriptor 2 (the standard error) to file descriptor 1 (the standard output).

<&- Close the standard input.

>&- Close the standard output.

Note that each command executed as a foreground process inherits the file descriptors and signal traps (described below) of the invoking shell, modified by any specified redirection. Background processes take input from the null device **/dev/null** (unless redirected), and ignore interrupt and quit signals.

File Name Patterns

The shell interprets an input token that contain any of the special characters '?', '*', or '[' as a file name *pattern*.

? Match any single character except newline. For example, the command

```
ls name?
```

will print the name of any file that consists of the string **name** plus any one character. If **name** is followed by no characters, or is followed by two or more characters, it will not be printed.

***** Match a string of non-newline characters of any length (including zero).

```
ls name*
```

will print the name of any file that begins with the string **name**, regardless of whether it is followed by any other characters. Likewise, the command

```
ls name?*
```

will print the name of any file that consists of the string **name** followed by at least one character. Unlike **name***, the token **name?*** insists that be followed by at least one character before it will be printed.

[!*xyz*]
Exclude characters *xyz* from the string search. For example, the command

```
ls [!abc]*
```

prints all files in the current directory except those that begin with **a**, **b**, or **c**.

[*C-c*]
Enclose alternatives to match a single character. A hyphen '-' indicates a range of characters. For example, the command

```
ls name[ABC]
```

will print the names of files **nameA**, **nameB**, and **nameC** (assuming, of course, that those files exist in the current directory). The command

```
ls name[A-K]
```

prints the names of files **nameA** through **nameK** (again, assuming that they exist in the current directory).

When **sh** reads a token that contains one of the above characters, it replaces the token in

the command line with an alphabetized list of file names that match the pattern. If it finds no matches, it passes the token unchanged to the command. For example, when you enter the command

```
ls name[ABC]
```

sh replaces the token **name[ABC]** with **nameA**, **nameB**, and **nameC** (again, if they exist in the current directory), so the command now reads:

```
ls nameA nameB nameC
```

It then passes this second, transformed version of the command line to the command **ls**.

Note that the slash '/' and leading period '.' must be matched explicitly in a pattern. The slash, of course, separates the elements of a path name; while a period at the begin of a file name usually (but not always) indicates that that file has special significance.

Quoting Text

From time to time, you will want to "turn off" the special meaning of characters. For example, you may wish to pass a token that contains a literal asterisk to a command; to do so, you need a way to tell **sh** not to expand the token into a list of file names. Therefore, **sh** includes the **quotation operators** '\', '"', and '"'; these "turn off" (or *quote*) the special meaning of operators.

The backslash '\' quotes the following character. For example, the command

```
ls name\*
```

lists a file named **name***, and no other.

The shell ignores a backslash immediately followed by a newline, called a *concealed newline*. This lets you give more arguments to a command than will fit on one line. For example, the command

```
cc -o output file1.c file2.c file3.c \
     file4.c file5.c file19.c
```

invokes the C compiler **cc** to compile a set of C source files, the names of which extend over more than one line of input. You will find this to be extremely helpful, especially when you write scripts and **makefiles**, to help you write neat, easily read commands.

A pair of apostrophes '' prevents interpretation of any enclosed special characters. For example, the command

```
find . -name '*.c' -print
```

finds and prints the name of any C-source file in the current directory and any subdirectory. The command **find** interprets the '*' internally; therefore, you want to suppress the shell's expansion of that operator, which is accomplished by enclosing that token between apostrophes.

A pair of quotation marks "" has the same effect. Unlike apostrophes, however, **sh** will perform parameter substitution and command-output substitution (described below) within quotation marks.

Scripts

Shell commands can be stored in a file, or *script*. The command

sh *script* [*parameter* ...]

executes the commands in *script* with a new subshell **sh**. Each *parameter* is a value for a positional parameter, as described below. If you have used the command **chmod** to make *script* executable, you may omit the **sh** command.

Parameters of the form '$*n*' represent command-line arguments within a script. *n* can range from zero through nine; **$0** always gives the name of the script. These parameters are also called *positional parameters*.

If no corresponding parameter is given on the command line, the shell substitutes the null string for that parameter. For example, if the script **format** contains the following line:

```
nroff -ms $1 >$1.out
```

then invoking **format** with the command line:

```
format mytext
```

invokes the command **nroff** to format the contents of **mytext**, and writes the output into file **mytext.out**. If, however, you invoke this command with the command line

```
format mytext yourtext
```

the script will format **mytext** but ignore **yourtext** altogether.

Reference **$*** represents all command-line arguments. If, for example, we change the contents of script **format** to read

```
nroff -ms $* >$1.out
```

then the command

```
format mytext yourtext
```

will invoke **nroff** to format the contents of **mytext** and **yourtext**, and write the output into file **mytext.out**.

Commands in a script can also be executed with the . (dot) command. It resembles the **sh** command, but the current shell executes the script commands without creating a new subshell or a new environment; therefore, you cannot use command-line arguments.

Variables

Shell variables are names that can be assigned string values on a command line, in the form

name=value

The name must begin with a letter, and can contain letters, digits, and underscores '_'. In shell input, '$*name*' or '${*name*}' represents the value of the variable. For example:

```
TEXT=mytext

nroff -ms $TEXT >$TEXT.out
```

Here, **sh** expands **$TEXT** before it executes the **nroff** command. This technique is very useful in large, complex scripts: by using variables, you can change the behavior of the script by editing one line, rather than having to edit numerous variables throughout the script.

Note that if an assignment precedes a command on the same command line, the effect of the assignment is local to that command; otherwise, the effect is permanent. For example,

```
kp=one testproc
```

assigns variable **kp** the value **one** only for the execution of the script **testproc**.

sh sets the following variables by default:

The number of actual positional parameters given to the current command.

@ The list of positional parameters "$1 $2 ...".

***** The list of positional parameters "$1" "$2" ... (the same as '$@' unless quoted).

- Options set in the invocation of the shell or by the **set** command.

? The exit status returned by the last command.

! The process number of the last command invoked with '&'.

$ The process number of the current shell.

sh also references the following variables:

CWD Current working directory: this is the name of the directory in which you are now working.

HOME Initial working directory; usually specified in the password file **/etc/passwd**.

IFS Delimiters for tokens; usually space, tab and newline.

LASTERROR Name of last command returning nonzero exit status.

MAIL Checked at the end of each command. If file specified in this variable is new since last command, the shell prints "You have mail." on the user's terminal.

PATH Colon-separated list of directories searched for commands.

PS1 First prompt string, usually '$'.

PS2 Second prompt string, usually '>'. **sh** prints it when it expects more input, such as when an open quotation-mark has been typed but a close quotation-mark has not been typed, or within a shell construct.

The special forms '${*nameCtoken*}' perform conditional parameter substition: *C* is one of the characters '-', '=', '+', or '?'. **sh** replaces the form '${*name-token*}' by the value of *name* if it is set, and by *token* otherwise. It handles the '=' form in the same way, but also sets the value of *name* to *token* if it was not set previously. **sh** replaces the '+' form by *token* if the given *name* is set. It replaces the '?' form by the value of *name* if set, and otherwise prints *token* and exits from the shell.

Command Output Substitution

sh can use the output of a command as shell input (as command arguments, for example) by enclosing the command in grave characters ` `. For example, to list the contents of the directories named in file **dirs**, use the command

```
ls -l `cat dirs`
```

Constructs

sh lets you control execution of commands by the constructs **break**, **case**, **continue**, **for**, **if**, **until**, and **while**. It recognizes each reserved word only if it occurs unquoted as the first token of a command. This implies that a separator must precede each reserved word in the following constructs; for example, newline or ';' must precede **do** in the **for** construct.

break *[n]*
Exit from **for**, **until**, or **while**. If *n* is given, exit from *n* levels.

case *token* **in** [*pattern* [| *pattern*] ...) *sequence*;;] ... **esac**
Check *token* against each *pattern*, and execute *sequence* associated with the first matching *pattern*.

continue *[n]*
Branch to the end of the *n*th enclosing **for**, **until**, or **while** construct.

for *name* [**in** *token* ...] **do** *sequence* **done**
Execute *sequence* once for each *token*. On each iteration, *name* takes the value of the next *token*. If the **in** clause is omitted, **$@** is assumed. For example, to list all files ending with **.c**:

```
for i in *.c
do
      cat $i
done
```

if *seq1* **then** *seq2* [**elif** *seq3* **then** *seq4*] ... [**else** *seq5*] **fi**
Execute *seq1*. If the exit status is zero, execute *seq2*; if not, execute the optional *seq3* if given. If the exit status of *seq3* is zero, then execute *seq4*, and so on. If the exit status of all tested sequences is nonzero, execute *seq5*.

until *sequence1* [**do** *sequence2*] **done**
Execute *sequence2* until the execution of *sequence1* results in an exit status of zero.

while *sequence1* [**do** *sequence2*] **done**
Execute *sequence2* as long as the execution of *sequence1* results in an exit status of zero.

(*sequence*
)
Execute *sequence* within a subshell. This allows *sequence* to change the current directory, for example, and not affect the enclosing environment. Note that the closing ')' must appear on the line that follows *sequence*.

{*sequence*
}
Braces simply enclose a *sequence*. Note that the closing '}' must appear on the line that follows *sequence*.

Special Commands

sh usually executes commands with the **fork** system call, which creates another process. However, **sh** executes the commands in this section either directly or with an **exec** system call. See the Lexicon articles on **fork()** and **exec** for details on these calls.

. *script* Read and execute commands from *script*. Positional parameters are not allowed. **sh** searches the directories named in the environmental variable **PATH** to find the given *script*.

: *[token ...]*
A colon ':' indicates a "partial comment". **sh** normally ignores all commands on a line that begins with a colon, except for redirection and such symbols as **$**, **{**, **?**, etc.

A complete comment: if **#** is the first character on a line, **sh** ignores all text that follows on that line.

cd *dir* Change the working directory to *dir*. If no argument is given, change to the home directory.

dirs **sh** lets you maintain a "directory stack", or stack of names of directories. You can push, pop, and otherwise manipulate the contents of this stack, which you can use for any purpose for which you need to access a number of directory names quickly. The command **dirs** prints the contents of the directory stack. The commands **pushd** and **popd** also manipulate the directory stack.

eval *[token ...]*
Evaluate each *token* and treat the result as shell input.

exec *[command]*
Execute *command* directly rather than performing **fork**. This terminates the current shell.

exit *[status]*
Set the exit status to *status*, if given; otherwise, the previous status is unchanged. If the shell is not interactive, terminate it.

export *[name ...]*
sh executes each command in an *environment*, which is essentially a set of shell variable names and corresponding string values. It inherits an environment when invoked, and normally it passes the same environment to each command it invokes. **export** specifies that the shell should pass the modified value of each given *name* to the environment of subsequent commands. When no *name* is given, **sh** prints the name and value of each variable marked for export.

popd *[N ...]*
Pop the directory stack. When used without an argument, it pops the stack once. When used with one or more numeric arguments, **popd** pops the specified items from the stack; item 0 is the top of the stack. (For information on the directory stack, see the entry for the command **dirs**, above.)

pushd *[dir0 ... dirN]*
Push *dir0* through *dirN* onto the directory stack, and change the current directory to the last directory pushed onto the stack. When called without an argument, **pushd** exchanges the two top stack elements. (For information on the directory stack, see the entry for the command **dirs**, above.)

read *name ...*
Read a line from the standard input and assign each token of the input to the corresponding shell variable *name*. If the input contains fewer tokens than the *name* list, assign the null string to extra variables. If the input contains more tokens, assign the last *name* the remainder of the input.

readonly *[name ...]*
Mark each shell variable *name* as a read only variable. Subsequent assignments to read only variables will not be permitted. With no arguments, print the name and value of each read only variable.

set [-ceiknstuvx *[name ...]]*
Set listed flag. If *name* list is provided, set shell variables *name* to values of positional parameters beginning with **$1**.

shift Rename positional parameter **1** to current value of **$2**, and so on.

times Print the total user and system times for all executed processes.

trap *[command] [n ...]*
Execute *command* if **sh** receives signal *n*. If *command* is omitted, reset traps to original values. To ignore a signal, pass null string as *command*. With *n* zero, execute *command* when the shell exits. With no arguments, print the current trap settings.

umask *[nnn]*
Set user file creation mask to *nnn*. If no argument is given, print the current file creation mask.

wait *[pid]*
Hold execution of further commands until process *pid* terminates. If *pid* is omitted, wait for all child processes. If no children are active, this command finishes immediately.

Command-line Options

-c *string*
Read shell commands from *string*.

-e Exit on any error (command not found or command returning nonzero status) if the shell is not interactive.

-i The shell is interactive, even if the terminal is not attached to it; print prompt strings. For a shell reading a script, ignore the signals **SIGTERM** and **SIGINT**.

-k Place all keyword arguments into the environment. Normally, **sh** places only assignments to variables preceding the command into the environment.

-n Read commands but do not execute them.

-s Read commands from the standard input and write shell output to the standard error.

-t Read and execute one command rather than the entire file.

-u If the actual value of a shell variable is blank, report an error rather than substituting the null string.

-v Print each line as it is read.

-x Print each command and its arguments as it is executed.

- Cancel the **-x** and **-v** options.

If the first character of argument 0 is '-', **sh** reads and executes the scripts **/etc/profile** and **$HOME/.profile** before reading the standard input. **/etc/profile** is a convenient place for initializing system-wide variables, such as **TIMEZONE**.

Files

/etc/profile — System-wide initial commands
$HOME/.profile — User-specific initial commands
/dev/null — For background input
/tmp/sh* — Temporary files

See Also

commands, dup(), environ, exec, fork(), ksh, login, newgrp, set, signal(), test

For a list of all commands associated with **sh**, see the section **Shell Commands** in the **commands** Lexicon article.

Introduction to sh, the Bourne Shell, tutorial

Diagnostics

sh notes on the standard error all syntax errors in commands, and all commands which it cannot find. Syntax errors cause a noninteractive shell to exit. It gives error messages if I/O redirection is incorrect. **sh** returns the exit status of the last command executed or the status specified by an **exit** command.

SHELL — Environmental Variable

Name the default shell
SHELL=*shell*

The environmental variable **SHELL** names the shell that COHERENT invokes when you log in. The default is **SHELL=/bin/sh**, which invokes the Bourne shell.

See Also

environmental variables, sh

shellsort() — General Function (libc)

Sort arrays in memory
void shellsort(*data*, *n*, *size*, *comp***)**
char ****data*; int** *n*, *size***; int (****comp***)();**

shellsort is a generalized algorithm for sorting arrays of data in memory, using D. L. Shell's sorting method. **shellsort** works with a sequential array of memory called *data*, which is divided into *n* parts of *size* bytes each. In practice, *data* is usually an array of pointers or structures, and *size* is the **sizeof** the pointer or structure.

Each routine compares pairs of items and exchanges them as required. The user-supplied routine to which *comp* points performs the comparison. It is called repeatedly, as follows:

```
(*comp)(p1, p2)
char *p1, *p2;
```

Here, *p1* and *p2* each point to a block of *size* bytes in the *data* array. In practice, they are usually pointers to pointers or pointers to structures. The comparison routine must return a negative, zero, or positive result, depending on whether *p1* is less than, equal to, or greater than *p2*, respectively.

Example

For an example of how to use this routine, see the entry for **string**.

See Also

ctype, general functions, qsort()
The Art of Computer Programming, vol. 3, pp. 84*ff*, 114*ff*

Notes

shellsort differs from the sort function **qsort** in that it uses an iterative algorithm that does not require much stack.

shift — Command

Shift positional parameters
shift

Commands to the shell can be stored in a file, or *script*. Positional parameters pass command-line variables to a script.

shift changes the values of positional parameters. The old parameter values **$2**, **$3**, ... become the new parameter values **$1**, **$2** .... **shift** also reduces the value of **$#**, which gives the number of positional parameters, by one.

The shell executes **shift** directly.

See Also

commands, ksh, sh

shm — Device Driver

Shared memory device driver

The device **/dev/shm** is an interface to the shared memory device driver. It is assigned major device 24 (minor device 0) and can be accessed as a character-special device.

Shared memory access operations are performed by seeks, reads, and writes through the interface **/dev/shm**. The desired seek location is (**shmid** << 16L) + offset.

Shared memory control operations are performed through the system call **ioctl**. The operations **shmctl** and **shmget** are performed with an integer parameter array. The first element of the array is reserved for the return value (default, -1). Subsequent elements represent arguments. **ioctl** passes **SHMCTL**, **SHMGET**, **SHMAT**, or **SHMDT** as the second argument, and the parameter array as the third argument. The first argument is an open file descriptor to **/dev/shm**. Seeks, reads, and writes on shared memory can be performed through the file descriptor **shmfd**.

Access

To access shared memory, do the following:

1. Be sure that **/dev/shm** is present as a special-character file with major number 24, minor number 0, and broad enough permissions. The command

   ```
   /etc/mknod /dev/shm c 24 0
   ```

   will create **/dev/shm** if it does not yet exist.

2. Become the superuser **root**. Execute the command

   ```
   /etc/drvld /drv/shm
   ```

   to load the driver.

3. Use the COHERENT system call **shmget()** to create a shared-memory segment and obtain **shmid** value for it.

4. Use the COHERENT system call **lseek()** to position for read or write of a shared-memory segment. The first argument to **lseek** is **shmfd**, which is an external declared in **<sys/shm.h>**. The second argument to **lseek** is a **long** whose high word is the segment identifier **shmid** and whose low word is the offset within the shared-memory segment. The third argument to **lseek** is zero.

5. Use the COHERENT system calls **read()** and **write()** to access the segment. Again, use **shmfd** as the file descriptor.

6. When you are finished using shared memory, use the call

```
shmctl(shmid, IPC_RMID, 0)
```

to remove segments when you are finished.

7. Finally, use **ps -d** to obtain the process identifier of the shared-memory driver. To unload the driver, become the superuser **root**, and then type the command

```
kill  -9 xxxx
```

where *xxxx* is the process identifier for the **shm** driver.

Note that this manner of proceding is not entirely in the spirit of System V IPC shared memory: COHERENT does not support functions **shmat()** and **shmdt()**. Unfortunately, true attachment of shared segments is not possible in SMALL-model systems.

Notes

If you allocate too many shared memory identifiers, you will exhaust kernel data space, and thus halt the system in its tracks.

Creating many large shared memory segments can exhaust main memory, as shared memory segments do not currently support swapping.

The functions **shmat** and **shmdt** are not currently supported.

Private shared memory is not supported. Shared memory segments must be removed manually when no longer required. To remove all shared memory segments use the following C code:

```
#include <sys/shm.h>

#define NSHMID 16

shmget( 0, 0, 0 );        /* must do first */

for ( id=0; id < NSHMID; ++id )
     shmctl( id, IPC_RMID, 0 );
```

To load **shm** into memory, use the command **drvld**.

Files

/usr/include/sys/ipc.h
/usr/include/sys/shm.h
/dev/shm
/drv/shm

See Also

device drivers, drvld, shmctl(), shmget()

shm.h — Header File

Definitions used with shared memory
#include <sys/shm.h>

shm.h defines constants and macros used by routines that implement the COHERENT shared-memory facility.

See Also

header files

shmctl() — COHERENT System Call

Control shared-memory operations

#include <sys/shm.h>
shmctl(*shmid*, *cmd*, *buf*)
int *shmid*, *cmd*; **struct shmid_ds** **buf*;

shmctl provides controls the COHERENT system's shared-memory facility. *cmd* specifies the operation to perform, as follows:

IPC_STAT Place the current value of each member of the data structure associated with *shmid* into the structure pointed to by *buf*.

IPC_SET Set the value of the following members of the data structure associated with *shmid* to the corresponding value found in the structure pointed to by *buf*:

```
shm_perm.uid
shm_perm.gid
shm_perm.mode /* only low 9 bits */
```

This *cmd* can be executed only by a process whose effective user ID equals either that of the superuser or **shm_perm.uid** in the data structure associated with *shmid*.

IPC_RMID Remove the system identifier specified by *shmid* from the system and destroy the shared memory segment and data structure associated with it. This *cmd* can be executed only by a process whose effective user ID equalS either that of the superuser or **shm_perm.uid** in the data structure associated with *shmid*.

shmctl fails if any of the following is true:

- *shmid* is not a valid shared memory identifier **shmget** sets **errno** to **EINVAL**.
- *cmd* is not a valid command (**EINVAL**).
- *cmd* equals **IPC_STAT** and operation permission is denied to the calling process (**EACCES**).
- *cmd* equals **IPC_RMID** or **IPC_SET** and the effective user identifier of the calling process does equals neither that of the superuser nor **shm_perm.uid** in the data structure associated with *shmid* (**EPERM**).
- *buf* points to an illegal address (**EFAULT**).

Return Value

Upon successful completion, **shmctl** returns zero; otherwise, it returns -1 and sets **errno** to an appropriate value.

Files

/usr/include/sys/ipc.h
/usr/include/sys/shm.h
/dev/shm
/drv/shm

See Also

COHERENT system calls, shm, shmget()

Notes

To improve portability, the COHERENT system implements its shared-memory functions as a device driver instead of as an actual system call.

shmget() — COHERENT System Call

Get shared-memory segment
#include <sys/shm.h>
shmget(*key*, *size*, *shmflg*)
key_t *key*; **int** *size*, *shmflg*;

shmget returns the shared-memory identifier associated with *key*.

A shared-memory identifier and associated data structure and shared memory segment of size *size* bytes is created for *key* if *key* does not already have a shared-memory identifier associated with it, and (*shmflg* & **IPC_CREAT**) is true.

Upon creation, the data structure associated with the new shared memory identifier is initialized as follows:

- **shm_perm.cuid**, **shm_perm.uid**, **shm_perm.cgid**, and **shm_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order nine bits of **shm_perm.mode** are set equal to the low-order nine bits of *shmflg*. These nine bits define access permissions: the top three bits give the owner's access permissions (read, write, execute), the middle three bits the owning group's access permissions, and the low three bits access permissions for others.
- **shm_segsz** is set equal to the value of *size*.
- **shm_lpid**, **shm_nattch**, **shm_atime**, and **shm_dtime** are set equal to zero. **shm_ctime** is set equal to the current time.

shmget fails if any of the following is true:

- *size* is less than the system-imposed minimum or greater than the system-imposed maximum. **shmget** sets **errno** to **EINVAL**.
- A shared-memory identifier exists for *key* but operation permission as specified by the low-order nine bits of *shmflg* would not be granted (**EACCES**).
- A shared-memory identifier exists for *key* but the size of the segment associated with it is less than *size* and *size* is not equal to zero (**EINVAL**).
- A shared-memory identifier does not exist for *key* and (*shmflg* & **IPC_CREAT**) is false (**ENOENT**).
- A shared-memory identifier is to be created but the system-imposed limit on the maximum number of allowed shared memory identifiers system-wide would be exceeded (**ENOSPC**).
- A shared-memory identifier and associated shared-memory segment are to be created, but the amount of available physical memory is not sufficient to fill the request (**ENOMEM**).
- A shared-memory identifier exists for *key* but ((*shmflg* & **IPC_CREAT**) **&&** (*shmflg* & **IPC_EXCL**)) is true (**EEXIST**).

Return Value

Upon successful completion, **shmget** returns a shared-memory identifier, which is always a non-negative integer. Otherwise, it returns -1 and sets **errno** to an appropriate value.

Files

/usr/include/sys/ipc.h
/usr/include/sys/shm.h
/dev/shm
/drv/shm

See Also

COHERENT system calls, shm, shmctl()

Notes

To improve portability, the COHERENT system implements its shared-memory functions as a device driver rather than actual system calls.

short — C Keyword

Data type

short is a numeric data type. By definition, it cannot be longer than an **int**. Under COHERENT, an **int** is equal to an **short**; that is, both **sizeof int** and **sizeof short** equals two **chars**, or 15 bits plus a sign. A **short** normally is sign extended when cast to a larger data type; however, an **unsigned short** will be zero extended when cast.

See Also

C keywords, data format, data type

shutdown — Command

Shut down the COHERENT system
/etc/shutdown

shutdown shuts down the COHERENT system. It is a shell script that leads you through each step of system shutdown. Only the superuser **root** can run **shutdown**. When shut down has been completed, the COHERENT system is in single-user mode. At this point, the user can safely run **fsck**, reboot the system, or turn the computer off.

Failure to shut down the system before rebooting or shutting off the computer could damage the COHERENT file system and destroy data.

See Also

commands, reboot

signal() — COHERENT System Call

Specify disposition of a signal
#include <signal.h>
int (*signal(*signum, action***))()**
int *signum*, **(***action***)();**

A process can receive a *signal*, or interrupt, from a hardware exception, terminal input, or a **kill** call made by another process. A hardware exception might be caused by an illegal instruction code or a bad machine address, caught by the segmentation hardware. A terminal interrupt character, described in detail in **tty**, generates a process interrupt (and in one case a core dump file for debugging purposes).

When a process receives a signal, it performs an appropriate *action*. The default action **SIG_DFL** causes the process to terminate. By calling signal, you can specify what action

the process takes when it receives a given signal *signum* is the number of the signal, and *action* points to the routine to execute when *signum* is received. The action **SIG_IGN** causes a signal to be ignored. Note that the signal **SIGKILL**, which kills a process, can be neither caught nor ignored. **signal** returns a pointer to the previous action.

With the exception of **SIGKILL** and **SIGTRAP**, caught signals are reset to the default action **SIG_DFL**. To catch a signal again, the specified *action* must reissue the **signal** call.

The following list gives machine-independent signals by symbolic name (defined in the header file **signal.h**), numeric value, and description. Signals marked by an asterisk produce a core dump if the action is **SIG_DFL**.

SIGHUP	1	Hangup
SIGINT	2	Interrupt
SIGQUIT	3*	Quit
SIGALRM	4	Alarm clock
SIGTERM	5	Termination
SIGREST	6	Restart indication
SIGSYS	7*	Bad system call argument
SIGPIPE	8	Write on closed pipe
SIGKILL	9	Kill
SIGTRAP	10*	Breakpoint
SIGSEGV	11*	Segmentation violation

The following lists gives machine-dependent signals defined in the header file **msig.h**.

The following signals are specific to the Zilog Z8002 version of COHERENT:

SIGUNI	12*	Unimplemented instruction
SIGPRV	13*	Privileged instruction
SIGNVI	14*	Non-vectored interrupt
SIGPAR	15*	Parity error

The following signals are specific to the Zilog Z8001 version of COHERENT:

SIGEPA	12*	Extended processor trap
SIGPRV	13*	Privileged instruction
SIGNVI	14*	Non-vectored interrupt
SIGNMI	15*	Non-maskable interrupt (not in all versions)

The following signals are specific to the Intel 8086 or 80286 version of COHERENT:

SIGDIVE	12*	Divide error
SIGOVFL	13*	Overflow

A signal may be caught during a system call that has not yet returned. In this case, the system call appears to fail, with **errno** set to **EINTR.** If desired, such an interrupted system call may be reissued. System calls which may be interrupted in this way include **pause**, **read** on a device such as a terminal, **write** on a pipe, and **wait**.

See Also

COHERENT system call, kill, ptrace(), sh, signame

Diagnostics

In case of an error, **signal** returns a pointer to a function returning **int**. That is, it returns **(int (*)())-1** for an invalid *signum*.

signal.h — Header Files

Declare signals
#include <signal.h>

The header file **signal.h** declares manifest constants that name all of the machine-independent signals that the COHERENT system uses to communicate with its processes. The header file **msig.h** declares constants for the machine-dependent signals.

See Also

header files, kill, msig.h, signal()

signame — Technical Information

Array of names of signals
#include <signal.h>
extern char *signame[*NSIG***+1];**

When a program terminates abnormally, its parent process receives a byte of termination information from the **wait** call. This byte contains a signal number, as defined in the header file **signal.h**. For example, **SIGINT** indicates an interrupt from the terminal.

The array **signame**, indexed by signal number, contains strings that give the meaning of each signal. Thus, **signame[SIGINT+1]** points to the string "interrupt". For portability reasons, all programs which wait on child processes (such as the shell **sh**) should use **signame**.

Files

<signal.h>

See Also

sh, signal(), technical information, wait

sin() — Mathematics function (libm)

Calculate sine
#include <math.h>
double sin(*radian***) double** *radian***;**

sin calculates the sine of its argument *radian*, which must be in radian measure.

Example

For an example of this function, see the entry for **acos**.

See Also

mathematics library

sinh() — Mathematics Function (libm)

Calculate hyperbolic sine
#include <math.h>
double sinh(*radian***) double** *radian***;**

sinh calculates the hyperbolic sine of *radian*, which is in radian measure.

Example

For an example of this function, see the entry for **cosh**.

See Also

mathematics library

size — Command

Print size of an object file
size [*file* ...]

size prints the sizes, in bytes, of the segments of each *file* (in decimal) and also prints the total size of all the segments (in both decimal and octal). Each *file* must be an object file.

One line is output for each file, listing the following segments:

Shared instructions
Private instructions
Uninitialized instructions
Shared data
Private data
Uninitialized data

If you specify the **-c** option, the total size of the common areas is displayed immediately after the uninitialized data.

See Also

commands, l.out.h

Notes

size makes no concessions to machines that use hexadecimal.

sizeof — C Keyword

Return size of a data element

sizeof is a C operator that returns a constant **int** that gives the size of any given data element. The element examined can be a data object, a portion of a data object, or a type cast. **sizeof** returns the size of the element in **chars**; for example

```
long foo;
sizeof foo;
```

returns four, because a **long** is as long as four **chars**.

sizeof can also tell you the size of an array. This is especially helpful for use with external arrays, whose size can be set when they are initialized. For example:

```
char *arrayname[] = {
     "COHERENT", "Mark Williams C for the Atari ST",
     "Let's C", "Fast Forward"
};

main()
{
     printf("\"arrayname\" has %d entries\n",
          sizeof(arrayname)/sizeof char*);
}
```

sizeof is especially useful in **malloc** routines, and when you need to specify byte counts to I/O routines. Using it to set the size of data types instead of using a predetermined value will increase the portability of your code.

See Also

C keywords, data types, operators

sleep() — General Function

Suspend execution for interval
sleep(*seconds***)**
unsigned *seconds***;**

sleep suspends execution for *seconds*.

Example

The following example, called **godot.c**, demonstrates how to use **sleep**.

```
main()
{
	printf("Waiting for Godot ...\n");

	for ( ; ; ) {
		/* sleep for five seconds */
		sleep(5);
		printf("... still waiting ...\n");
	}
}
```

See Also

general functions

sleep — Command

Stop executing for a specified time
sleep *seconds*

The command **sleep** suspends execution for a specified number of *seconds*. This routine is especially useful with other commands to the shell. For example, typing

```
	(sleep 3600; echo coffee break time) &
```

executes the **echo** command in one hour (3,600 seconds) to indicate an important appointment.

See Also

alarm(), commands, ksh, pause(), sh

sload() — COHERENT System Call

Load device driver
#include <con.h>
int sload(*major, file, conp***)**
int *major***; char *** *file***; CON *** *conp***;**

The COHERENT system accesses all devices through drivers residing in the system. Except for the root device, drivers must be explicitly loaded before use; this operation does not involve re-booting.

sload loads the driver given by *file* as device number *major*. This number uniquely identifies the driver to the system. *conp* is a reference to a **CON** structure, as defined in the header file **con.h**. It describes standard entry points and gives other information on the driver. Normally, *major* and *conp* are obtained from the driver load module; this is the method used

by the **load** command.

file must be in the correct format. Usually, it is created using the **-k** option to **ld**.

This call is restricted to the superuser.

Files

<con.h>
/drv/*

See Also

COHERENT system calls, con.h, init, l.out.h, ld, suload

Diagnostics

sload return zero upon successful loading of the appropriate driver, or -1 on errors. **sload** errors include nonexistent *file*, parameter (such as *major*) out of range, driver already loaded for *major*, or *file* not a file containing a proper driver.

smult() — Multiple-Precision Mathematics

Multiply multiple-precision integers
#include <mprec.h>
void smult(*a*, *n*, *c*)
mint **a*, **c*; **int** *n*;

The COHERENT system includes a suite of routines that allow you to perform multiple-precision mathematics. The function **smult** multiplies the multiple-precision integer (or **mint**) pointed to by *a* by the integer *n*, which is <= 127. It writes the product into the **mint** pointed to by *c*.

See Also

multiple-precision mathematics

sort — Command

Sort lines of text
sort [-bcdfimnru] [-t *c*] **[-o** *outfile*] **[-T** *dir*] **[+***beg***[-***end***]][***file* ...]

sort reads lines from each *file*, or from the standard input if no file is specified. It sorts what it reads, and writes the sorted material to the standard output.

sort sorts lines by comparing a *key* from each line. By default, the key is the entire input line (or *record*) and ordering is in ASCII order. The key, however, can be one or more *fields* within the input record; by using the appropriate options, you can select which fields are used as the key, and dictate the character that is used to separate the fields.

The following options affect how the key is constructed or how the output is ordered.

-b Ignore leading white space (blanks or tabs) in key comparisons.

-d Dictionary ordering; use only letters, blanks, and digits when comparing keys. This is essentially the ordering used to sort telephone directories.

-f Fold upper-case letters to lower case for comparison purposes.

-i Ignore all characters outside of the printable ASCII range (octal 040-0176).

-n The key is a numeric string that consists of optional leading blanks and optional minus sign followed by any number of digits with an optional decimal point. Ordering is by the numeric, as opposed to alphabetic, value of the string.

-r Reverse the ordering, i.e., **sort** from largest to smallest.

As noted above, the key compared from each line need not be the entire input line. The option *+beg* indicates the beginning position of the key field in the input line, and the optional *-end* indicates that the key field ends just before the *end* position. If no *-end* is given, the key field ends at the end of the line. Each of these positional indicators has the form *+m.nf* or *-m.nf*, where *m* is the number of fields to skip in the input line and *n* is the number of characters to skip after skipping fields. Optional flags *f* are chosen from the above key flags (**bdfinr**) and are local to the specified field.

The following additional options control how **sort** works.

-c Check the input to see if it is sorted. Print the first out-of-order line found.

-m Merge the input files. **sort** assumes each *file* to be sorted already. With large files, **sort** runs much faster with this option.

-o **outfile**
Put the output into *outfile* rather than on the standard output. This allows **sort** to work correctly if the output file is one of the input files.

-tc Use the character *c* to separate fields rather than the default blanks and tabs. For example, **-t/** uses the slash instead of white space to separate fields; this is useful when sorting file names and directory names.

-T **dir**
Create temporary files in directory *dir* rather than the standard place.

-u Suppress multiple copies of lines with key fields that compare equally.

The following example sorts the password file **/etc/passwd**, first by group number (field 4) and then by user name (field 1):

```
sort -t: +3n -4 +0 -1 /etc/passwd
```

Files

/usr/tmp/sort* — First attempt at temporary files
/tmp/sort* — Second attempt at temporary files

See Also

ASCII, commands, ctype, tsort, uniq

Diagnostics

sort returns a nonzero exit status if internal problems occurred or if the file was not correctly sorted, in the case of the **-c** option.

spell — Command

Find spelling errors
spell [-a][-b][*file* ...]

spell builds a set of unique words from a document contained in each input *file*, or the standard input if none. It writes a list of words believed to be misspelled onto the standard output.

spell should normally be invoked with the document in the form of the input to the text formatter **nroff** rather than the output. **spell** deletes control information to the formatter by invoking **deroff**.

The default dictionary is for American spelling of English. The **-a** option specifies this dictionary explicitly. Under the **-b** option, British spelling is checked. This accepts *favour*,

fibre, and *travelled* rather than the American spellings *favor*, *fiber*, and *traveled* for the same words. Words ending in *ize* are also accepted when ending in *ise* (e.g., *digitize*, *digitise*).

The dictionary has a reasonably complete coverage of proper names as well as technical terms in certain fields. However, it covers some fields (e.g., computer science) better than others (e.g., medicine).

Files

/usr/dict/clista — Compressed American dictionary
/usr/dict/clistb — Compressed British dictionary
/usr/dict/spellhist — History file for dictionary maintainer
/usr/lib/spell

See Also

commands, deroff, nroff, sort, typo

Notes

Dictionaries are not provided for languages other than English.

No dictionary can be complete. You must add new words to the dictionary to ensure that it fully meets your needs.

Obscure words (such as opcodes, variable names, etc.) are flagged as spelling errors.

Because the data files required for **spell** are quite large, they might not be included on COHERENT systems for machines with insufficient disk space. As a result, the command might not work as expected on all systems.

split — Command

Split a text file into smaller files
split [*-nlines*][**-c***count*][*infile* [*outfile*]]

split divides a text file into a number of smaller files. This is useful for programs that cannot handle arbitrarily large files, such as MicroEMACS.

split uses *infile* as its input file if given; otherwise, it uses the standard input. If *infile* is '-', **split** uses the standard input.

split puts its output into files with names prefixed by *outfile* and suffixed consecutively with **aa**, **ab**, **ac**, and so on. If no *outfile* is specified, file names are prefixed with **x**.

Normally, **split** puts 1,000 lines in each output file. This default may be changed for text files by the option *-nlines*, where *nlines* gives the desired number of lines per file. When using **split** on binary files, the *count* argument to the **-c** option allows you to specify the number of characters to place in each output file.

See Also

commands

spow() — Multiple-Precision Mathematics

Raise multiple-precision integer to power
#include <mprec.h>
void spow(*a*, *n*, *b*)
mint *****a*, *****b*; **int** *n*;

The COHERENT system includes a suite of routines that allow you to perform multiple-precision mathematics. The function **spow** raises the multiple-precision integer (or **mint**) pointed to by *a* to the power of *n*, and writes the result into the **mint** pointed to by *b*. In no case may the exponent be negative.

LEXICON

See Also

multiple-precision mathematics

sprintf() — STDIO (libc)

Format output
#include <stdio.h>
int sprintf(*string*, *format* [, *arg*] ...**)**
**char ****string*, **format*;

sprintf formats and prints a string. It resembles the function **printf**, except that it writes its output into the memory location pointed to by **string**, instead of to the standard output.

sprintf reads the string pointed to by *format* to specify an output format for each *arg*; it then writes every *arg* into *string*, which it ends with a null character. For a detailed discussion of **sprintf**'s formatting codes, see **printf**.

Example

For an example of this function, see the entry for **sscanf**.

See Also

printf(), fprintf(), STDIO

Notes

The output *string* passed to **sprintf** must be large enough to hold all output characters.

Because C does not perform type checking, it is essential that each argument match its format specification.

At present, **sprintf** does not return a meaningful value.

sqrt() — Mathematics Function (libm)

Compute square root
#include <math.h>
double sqrt(*z***) double** *z*;

sqrt returns the square root of *z*.

Example

For an example of this function, see the entry for **ceil**.

See Also

mathematics library

Diagnostics

When a domain error occurs (i.e., when *z* is negative), **sqrt** sets **errno** to **EDOM** and returns zero.

srand() — General Function (libc)

Seed random number generator
void srand(*seed***) int** *seed*;

srand uses *seed* to initialize the sequence of pseudo-random numbers returned by **rand**. Different values of *seed* initialize different sequences.

Example

For an example of this function, see the entry for **rand**.

See Also

general functions, rand()
The Art of Computer Programming, vol. 2

srcpath — Command

Find source files
srcpath [-aw] [-p *path]filename pattern* **...**

The command **srcpath** expands the environmental variable **SRCPATH**, applies it to each argument, and prints the full path of each unique result.

An argument can either be a file name or a pattern. For example, the command

```
srcpath "*.[ch]"
```

finds all **.c** and **.h** files on **SRCPATH**. By default, **srcpath** keeps only the first file that it finds with a given name. **srcpath** automatically appends '.' to the beginning of **SRCPATH** so files in the current directory have precedence.

srcpath recognizes the following command-line options:

-p *path*
Use *path* as its path instead of **SRCPATH**. For example,

```
srcpath -p ".:/usr/src/cmd" "*.c"
```

tells **srcpath** to search '.' and **/usr/src/cmd** instead of **SRCPATH**. Note that with this option, **srcpath** does not automatically place '.' at the beginning of the list.

-a Disable shadowing. Normally, if **srcpath** finds a file is found in more than one directory on the path, it prints only the first. The **-a** option forces **srcpath** to print all instances of the file name.

-w By default, **srcpath** silently bypasses directories and matching files for which it has no read permission. The **-w** option causes it to print a warning message when this happens.

See Also

commands, find, make, PATH

ss — Device Driver

Future Domain/Seagate SCSI device driver

The device driver **ss** lets you use SCSI interface devices attached to any of the following host adapters:

Future Domain TMC-845/850/860/875/885
Future Domain TMC-840/841/880/881
Seagate ST01/ST02

This driver has major number 13. It can be accessed either as a block-special device or as a character-special device. The minor number specifies the device and partition number for disk-type devices; this lets you use up to eight SCSI-IDs, with one logical unit number (LUN) per SCSI-ID and up to four partitions per LUN. The present version supports only LUN 0 per SCSI-ID.

The first **open** call on a SCSI disk device reads the partition table into memory.

Controller Configuration

For your Future Domain or Seagate host adapter to work with COHERENT, you must install it with interrupts enabled. If you have been running your host adapter with interrupts disabled, a good first choice for interrupt number is IRQ 5, unless you know that you have another device installed on your computer that already uses this interrupt. Consult the instructions provided with your host adapter, and the jumper settings, to determine the IRQ number.

The base address value used by **ss** is the four-digit hexadecimal memory-segment number of the host adapter's starting address. This number is most often **CA00**; other common values are **C800**, **CC00**, **CE00**, **DC00**, and **DE00**. You must use the correct value, as specified by the jumper settings on your host adapter.

Device driver variables **SS_BASE_** and **SS_INT_** correspond to the base address and interrupt vector, respectively. Device driver variable **NSDRIVE_** must be patched before the driver is loaded. The low-order byte of this variable is a bit map that indicates the SCSI-IDs of all installed target devices. The high-order byte indicates the type of host adapter. Labelling the bits in the low-order byte of **NSDRIVE_** is as follows:

Bit number: 7 6 5 4 3 2 1 0 ← *least significant bit*

There should be a value of one for each installed target device. Do not set a value of one for the SCSI-ID of the host adapter. The high-order byte of **NSDRIVE_** is **0x00** for Seagate ST01 and ST02, **0x80** for TMC-845/850/860/875/885, and **0x40** for TMC-840/841/880/881. For example, if you are using a TMC-885 and a single hard drive with SCSI ID of zero, then set **NSDRIVE_** to 0x8001. See Lexicon article **hs** for an example of how to configure a device driver.

When processing BIOS I/O requests prior to booting COHERENT, SCSI host adapters use "translation-mode" drive parameters: number of heads, cylinders, and sectors per track. These numbers are called translation-mode parameters because they have nothing to do with physical drive geometry. The translation-mode parameters used by the BIOS code present on your host adapter can be obtained using the **dpb** utility found on the boot diskette of versions 3.2.0 and later of COHERENT.

ss has a table, **drv_parm_**, which contains eight two-word entries, one for each possible SCSI-ID. The first word of each entry must contain the number of cylinders for the drive. The high-order byte of the second word is the number of sectors per track; the low-order byte is the number of heads. Entries in **drv_parm_** should be patched for each drive which is accessible by the BIOS. Values need not be patched for drives inaccessible by the BIOS. Note that BIOS code is executed by COHERENT only during the initial bootstrap. After that, drive parameters are of no consequence because SCSI I/O requests are based upon logical block number, rather than upon cylinder/head/sector addressing.

The installation procedure for COHERENT versions 3.2.0 and later patches all necessary variables for the accompanying version of the **ss** driver by executing the command:

```
/etc/mkdev scsi
```

Minor Device Numbers

ss usually uses the special files **/dev/sd*** and **/dev/rsd***. For information on the meaning of minor numbers with these special files, see the article on **aha154x**.

Loading the Driver

ss must be loaded on a system that does not have a SCSI hard disk as the root device. To do so, use the command **/etc/drvld**, as follows:

```
/etc/drvld -r /drv/ss
```

Files

/dev/sd* — block-special devices
/dev/rsd* — character-special devices

See Also

device drivers, drvld, scsi

Notes

Current releases of **ss** support disk-type devices only. Zero is the only LUN allowed. Future versions will add support for tape-type and other devices, as well as nonzero LUN's.

In version 3.2.0 of COHERENT, another variable, **SS_HOST_**, must be patched in the driver to be equal to the SCSI-ID of the host adapter. This value is six for Future Domain adapters, and seven for Seagate. Variable **SS_HOST_** has been deleted from versions of the **ss** driver later than that shipped with COHERENT 3.2.0.

sscanf() — STDIO (libc)

Format a string
#include <stdio.h>
int sscanf(*string*, *format* [, *arg*] ...)
**char ***string*; **char ***format*;

sscanf reads the argument *string*, and uses *format* to specify a format for each *arg*, each of which must be a pointer. For more information on **sscanf**'s conversion codes, see **scanf**.

Example

This example uses **sprintf** to create a string, and then reads it with **sscanf**. It also illustrates a common problem with this routine.

```
#include <stdio.h>

main()
{
	char string[80];
	char s1[10], s2[10];

	sprintf(string, "1234567890123456789012345678 90");
	sscanf(string, "%9c", s1);
	sscanf(string, "%10c", s2);

	printf("\n%s is the string\n", string);
	printf("%s: first 9 characters in string\n", s1);
	printf("%s: first 19 characters in string\n", s2);
}
```

See Also

fscanf(), scanf(), STDIO

Diagnostics

sscanf returns the number of arguments filled. It returns zero if no arguments can be filled or if an error occurs.

Notes

Because C does not perform type checking, an argument must match its format specification. **sscanf** is best used only to process data that you are certain are in the correct data format, such as data that were written with **sprintf**.

sscanf is difficult to use correctly, and incorrect usage can create serious bugs in programs. It is recommended that **strtok** be used instead.

stack — Definition

The **stack** is the segment of memory that holds function arguments, local variables, function return addresses, and stack frame linkage information. The COHERENT library sets the stack size to four kilobytes. You can change the size of the stack by using the command **fixstack**.

If your program uses recursive algorithms, or declares large amounts of automatic data, or simply contains many levels of functions calls, the stack may "overflow", and overwrite the program data.

See Also

definitions, fixstack

standard error — Definition

The **standard error** is the peripheral device or file where programs write error messages by default. It is defined in the header file **stdio.h** under the abbreviation **stderr**, and by default is the computer's monitor.

The COHERENT shell **sh** lets you redirect into a file all text written to the standard error device. To do so, use the shell operator **2>**. For example

```
make 2>errorfile
```

redirects all error messages generated by **make** into file **errorfile**.

See Also

definitions, stderr, stdio.h

standard input — Definition

The **standard input** is the device or file from which data are accepted by default. It is defined in the header file **stdio.h** under the abbreviation **stdin**, and will be the computer's keyboard unless redirected by the operating system, a shell, or **freopen**.

The COHERENT shell **sh** lets you redirect the standard input device. To do so, use the shell operator **<**. For example

```
mail fwb <textfile
```

the standard input device from your terminal to file **textfile**; in effect, this commands mails the contents of **textfile** to user **fwb**.

See Also

definitions, stdin, stdio.h

standard output — Definition

The **standard output** is the device or file where programs write output by default. It is defined in the header file **stdio.h** under the abbreviation **stdout**, and in most instances is defined to be the computer's monitor.

The COHERENT shell **sh** lets you redirect into a file all text written to the standard output device. To do so, use the shell operator **>**. For example

```
sort myfile >sortfile
```

redirects the text output by **sort** into file **sortfile**.

See Also

definitions, stdio.h, stdout

stat() — General Function (libc)

Find file attributes
#include <sys/stat.h>
int stat(*file*, *statptr***)**
char ****file*; struct stat ****statptr***;**

stat returns a structure that contains the attributes of a file, including protection information, file type, and file size.

file points to the path name of file. *statptr* points to a structure of the type **stat**, as defined in the header file **stat.h**.

The following summarizes the structure **stat**:

```
struct stat {
      dev_t   st_dev;       /* Device */
      ino_t   st_ino;       /* i-node number */
      unsigned short st_mode;/* Mode */
      short   st_nlink;     /* Link count */
      short   st_uid;       /* User id */
      short   st_gid;       /* Group id */
      dev_t   st_rdev;      /* Real device */
      fsize_t st_size;      /* Size */
      time_t  st_atime;     /* Access time */
      time_t  st_mtime;     /* Modify time */
      time_t  st_ctime;     /* Change time */
};
```

The following lists the legal settings for the element **st_mode** which defines the file's attributes:

S_IFMT	0x0170000	File types
S_IFREG	0x0100000	Ordinary file
S_IFDIR	0x0040000	Directory
S_IFCHR	0x0020000	Character-special file
S_IFBLK	0x0060000	Block-special file
S_ISUID	0x0004000	Set user identifier
S_ISGID	0x0002000	Set group identifier
S_ISVTX	0x0001000	Save text bit
S_IREAD	0x0000400	Owner read permission
S_IWRITE	0x0000200	Owner write permission
S_IEXEC	0x0000100	Owner execute permission

st_dev and **st_ino** together form a unique description of the file. The former is the device on which the file and its i-node reside, and the latter is the index number of the file. **st_mode** gives the permission bits, as outlined above. **st_nlink** gives the number of links to the file. The user id and group id of the owner are **st_uid** and **st_gid,** respectively. **st_rdev**, which is valid only for special files, holds the major and minor numbers for the file.

The entry **st_size** gives the size of the file, in bytes. For a pipe, the size is the number of bytes waiting to be read from the pipe.

Three entries for each file give the last occurrences of various events in the file's history. **st_atime** gives the time the file was last read or written to. **st_mtime** gives the time of the last modification, write for files, create or delete entry for directories. **st_ctime** gives the last change to the attributes, not including times and size.

Example

The following example uses **stat** to print a file's status.

```
#include <sys/stat.h>
main()
{
     struct stat sbuf;
     int status;

     if (status = stat("/usr/include", &sbuf)) {
          printf("Can't find\n");
          exit(1);
     }

     printf("uid = %d gid = %d\n", sbuf.st_uid, sbuf.st_gid);
}
```

Files

<sys/stat.h>

See Also

chmod(), chown(), COHERENT system calls, ls, open()

Notes

stat differs from the related function **fstat** mainly in that **fstat** accesses the file through its descriptor, which was returned by a successful call to **open**, whereas **stat** takes the file's path name and opens it before checking its status.

Diagnostics

stat returns -1 if an error occurs, e.g., the file cannot be found. Otherwise, it returns zero.

stat.h — Header File

Definitions and declarations used to obtain file status
#include <sys/stat.h>

stat.h is a header file that contains the declarations of several structures used by the routines **fstat** and **stat**, which return information about a file's status.

See Also

chmod(), fstat(), header file, stat()

static — C Keyword

Declare storage class

static is a C storage class. It has two entirely different meanings, depending upon whether it it appears inside or outside a function.

Outside a function, **static** means that the function or variable it preceeds may not be seen outside the module.

Inside a function, static may only precede a variable. It means that that variable is permanently allocated, rather than allocated on the stack when the function is entered and discarded when the function exits. If a **static** variable is initialized, that occurs before the program starts rather than every time the function is entered. If a function returns a pointer to a variable, often that variable is declared **static** within the function. If a pointer to a **non-static** local variable is returned, that variable is freed when the function returns and the pointer points to an unprotected location.

Example

The following example demonstrates the uses of the **static** keyword. It returns the next integer in a sequence as a string.

```
/* static to keep function hidden outside of this module */
static char *nextInt()
{
     /* static to protect value between calls */
     static int next = 0;
     /* static to allow the return of a pointer to s */
     static char s[5];

     sprintf(s, "%d", next++);
     return(s);
}
```

See Also

auto, C keywords, extern, register variable, storage class

stdarg.h — Header File

Header for variable numbers of arguments
#include <stdarg.h>

The header **stdarg.h** declares and defines the routines used to traverse a variable-length argument list. It declares the type **va_list** and the function **va_end**, and it defines the macros **va_start** and **va_arg**.

See Also

header files, varargs.h

stddef.h — Header File

Header for standard definitions
#include <stddef.h>

stddef.h defines three types and two macros that are used through the library. They are as follows:

NULL	Null pointer
offsetof()	Offset of a field within a structure
ptrdiff_t	Numeric difference between two pointers
size_t	Type returned by **sizeof** operator
wchar_t	Typedef for wide **chars**

See Also

header files

stderr — Definition

stderr is the name of the **FILE** pointer assigned to the standard error device. It is set in the header file **stdio.h**.

See Also

definitions, stdin, stdio.h, stdout, standard error

stdin — Definition

stdin is the name of the **FILE** pointer that is assigned to the standard input device. It is set in the header file **stdio.h**.

See Also

definitions, standard input, stderr, stdio.h, stdout

STDIO — Overview

STDIO is an abbreviation for *standard input and output*. It refers to a set of standard library functions that accompany all C compilers and that govern input and output with peripheral devices.

COHERENT includes the following STDIO routines:

clearerr() . . . . . Present status stream
fclose() . . . . . Close a file stream
fdopen() . . . . . Open a file stream for I/O
feof() . . . . . Discover a file stream's status
ferror() . . . . . Discover a file stream's status
fflush() . . . . . Flush an output buffer
fgetc(). . . . . Get a character
fgets(). . . . . Get a string
fgetw() . . . . . Get a word
fileno() . . . . . Get a file descriptor
fopen() . . . . . Open a file stream
fprintf(). . . . . Format and print to a file stream
fputc() . . . . . Output a character
fputs() . . . . . Output a string
fputw() . . . . . Output a word

fread() . . . . . Read a file stream
freopen() . . . . . Open a file stream
fscanf() . . . . . Format and read from a file stream
fseek() . . . . . Seek in a file stream
ftell() . . . . . Return file pointer position
fwrite() . . . . . Write to a file stream
getc() . . . . . Get a character
getchar() . . . . . Get a character
gets() . . . . . Get a string
getw() . . . . . Get a word
pclose() . . . . . Close a pipe
popen() . . . . . Open a pipe
printf() . . . . . Print a formatted string
putc() . . . . . Output a character
putchar() . . . . . Output a character
puts() . . . . . Output a string
putw() . . . . . Output a word
rewind() . . . . . Reset a file pointer
scanf() . . . . . Format and input from standard input
setbuf() . . . . . Set alternative file-stream buffers
sprintf() . . . . . Format and print to a string
sscanf() . . . . . Format and read from a string
ungetc() . . . . . Return character to file stream

STDIO routines are buffered by default.

See Also

buffer, FILE, Libraries, stdio.h, stream

stdio.h — Header File

Declarations and definitions for I/O

stdio.h is a header file that defines several manifest constants used in standard I/O, such as **NULL** and **FILE**, declares the STDIO functions, and defines numerous I/O macros.

See Also

header file, STDIO

stdlib.h — Header File

Declare/define general functions
#include <stdlib.h>

stdlib.h is a header file that is defined in the ANSI standard. It declares a set of general utilities and defines attending macros and data types, as follows:

Types

div_t . . . . . Type of object returned by **div**
ldiv_t . . . . . Type of object returned by **ldiv**

Manifest Constants

EXIT_FAILURE . . Value to indicate that program failed to execute properly
EXIT_SUCCESS. . Value to indicate that program executed properly
MB_CUR_MAX . . Largest size of multibyte character in current locale
RAND_MAX . . . . Largest size of pseudo-random number

Functions

abort() . . . . . End program immediately
abs(). . . . . Compute the absolute value of an integer
atof() . . . . . Convert string to floating-point number
atoi() . . . . . Convert string to integer
atol() . . . . . Convert string to long integer
bsearch(). . . . . Search an array
calloc() . . . . . Allocate dynamic memory
div(). . . . . Perform integer division
exit() . . . . . Terminate a program gracefully
free() . . . . . De-allocate dynamic memory to free memory pool
getenv() . . . . . Read environmental variable
labs() . . . . . Compute the absolute value of a long integer
ldiv() . . . . . Perform long integer division
malloc() . . . . . Allocate dynamic memory
qsort() . . . . . Sort an array
rand() . . . . . Generate pseudo-random numbers
realloc() . . . . . Reallocate dynamic memory
srand() . . . . . Seed the random-number generator
strtod(). . . . . Convert string to floating-point number
strtol() . . . . . Convert string to long integer
strtoul() . . . . . Convert string to unsigned long integer
system() . . . . . Suspend a program and execute another

See Also

header files

stdout — Definition

stdout is the name of the **FILE** pointer that is assigned to the standard output device. It is set in the header file **stdio.h**.

See Also

definitions, standard output, stderr, stdin, stdio.h

sticky bit — Definition

The *sticky bit* is one of the mode bits associated with a file. If the sticky bit is set for an executable file and swapping is enabled, COHERENT behaves in a special way when it executes that file.

When the COHERENT system executes the file the first time, all proceeds normally. When the program exits, however, the pure segments are left on the swap device; when the program is re-invoked, COHERENT reads "pure" code (text) areas from the swap device and all other (impure) segments from the file system. This speeds execution of large programs that are executed frequently.

This strategy works well on systems that have large swap devices. Because overuse of the sticky bit would quickly swamp the swap device, only the superuser can set the sticky bit.

See Also

chmod, definitions

stime() — COHERENT System Call

Set the time
#include
int stime(*timep***)**
time_t ***timep*;**

stime sets the system time. *timep* points to a variable of type **time_t**, which contains the number of seconds since midnight GMT of January 1, 1970.

stime is restricted to the superuser.

Files

<sys/types.h>

See Also

COHERENT system calls, ctime(), date, ftime(), stat(), utime()

Diagnostics

stime returns -1 on error, zero otherwise.

storage class — Technical Information

Storage class refers to the part of a declaration that indicates how data are to be stored. The C language recognizes the following storage clases:

auto
extern
register
static

typedef is technically defined as a storage class as well, but it does not actually indicate how data are stored. The default class is **auto**.

See Also

auto, extern, register, static, technical information, typedef

strcat() — String Function (libc)

Concatenate strings
#include <string.h>
char *strcat(*string1***, ***string2***)**
char ***string1*, ***string2***;**

strcat appends all characters in *string2* onto the end of *string1*. It returns the modified *string1*.

Example

For an example of this function, see the entry for **string functions**.

See Also

string functions, string.h, strncat()

Notes

string1 must point to enough space to hold itself and *string2*; otherwise, another portion of the program may be overwritten.

strchr() — String Function (libc)

Find a character in a string
#include <string.h>
char *strchr(*string*, *character***)**
char ***string*; int** *character***;**

strchr searches for the first occurrence of *character* within *string*. The null character at the end of *string* is included within the search. It is equivalent to the COHERENT function **index**.

strchr returns a pointer to the first occurrence of *character* within *string*. If *character* is not found, it returns NULL.

Having **strchr** search for a null character will always produce a pointer to the end of a string. For example,

```
char *string;
assert(strchr(string, '\0') == string + strlen(string));
```

never fails.

See Also

string functions

strcmp() — String Function (libc)

Compare two strings
#include <string.h>
int strcmp(*string1*, *string2***)**
char ***string1*, ***string2*;**

strcmp compares *string1* with *string2* lexicographically. It returns zero if the strings are identical, returns a number less than zero if *string1* occurs earlier alphabetically than *string2*, and returns a number greater than zero if it occurs later. This routine is compatible with the ordering routine needed by **qsort**.

Example

For examples of this function, see the entries for **string functions** and **malloc**.

See Also

qsort(), shellsort(), string functions, string.h, strncmp()

strcoll() — String Function (libc)

Compare two strings, using locale-specific information
#include <string.h>
int strcoll(*string1*, *string2***)**
char ***string1*; char ***string2***;**

strcoll lexicographically compares the string pointed to by *string1* with one pointed to by *string2*. Comparison ends when a null character is read.

strcoll compares the two strings character by character until it finds a pair of characters that are not identical. It returns a number less than zero if the character in *string1* is less (i.e., occurs earlier in the character table) than its counterpart in *string2*. It returns a number greater than zero if the character in *string1* is greater (i.e., occurs later in the character table) than its counterpart in *string2*. If no characters are found to differ, then the strings are identical and **strcoll** returns zero.

See Also

string functions, string.h

Notes

The string-comparison routines **strcoll**, **strcmp**, and **strncmp** differ from the memory-comparison routine **memcmp** in that they compare strings rather than regions of memory. They stop when they encounter a null character, but **memcmp** does not.

The ANSI Standard's description of **strcoll** emphasizes that it uses locale-specific information, as set by the ANSI function **setlocale**, to perform string comparisons. The COHERENT system has not yet implement ANSI locales; therefore, **strcoll** does not differ significantly from **strcmp**. It is included to support programs written in ANSI C.

strcpy() — String Function (libc)

Copy one string into another

#include <string.h>
char *strcpy(*string1*, *string2***)**
char ***string1*, ***string2*;**

strcpy copies the contents of *string2*, up to the null character, into *string1* and returns *string1*.

Example

See **string**.

See Also

memcpy(), string functions, string.h, strncpy()

Notes

string1 must point to enough space to hold *string2*, or another portion of the program or operating system may be overwritten.

strcspn() — String Function (libc)

Return length a string excludes characters in another

#include <string.h>
unsigned int strcspn(*string1*, *string2***)**
char ***string1*, ***string2*;**

strcspn compares *string1* with *string2*. It then returns the length, in characters, for which *string1* consists of characters *not* found in *string2*.

See Also

string functions, string.h

stream — Definition

The term **stream** is a metaphor for any entity that can be named and from which bits can flow, such as a device or a file. The name "stream" reflects the fact that the C programming environment does not depend upon record descriptors and other devices that predetermine what form data can assume; instead, data from whatever source are conceived as being a flow of bytes whose significance is set entirely by the program that reads them.

For example, whether 16 bits forms an **int**, two **char**s, and should be used as an absolute value or a bit map, is entirely up to the program that receives it. It is also irrelevant to the program that processes these 16 bits whether they come from the keyboard, from a file on disk, or from a peripheral device.

The **FILE** structure holds all of the information needed to manipulate a stream. The STDIO functions can be used to open, close, or reopen a stream; read data from it; or write data to it.

See Also

bit, byte, data formats, definitions, file, FILE, STDIO

stream.h — Header File

Definitions for message facility
#include <stream.h>

stream.h definitions constants and structures used by the routines that implement the COHERENT message facility.

See Also

header files

strerror() — String Function (libc)

Translate an error number into a string
#include <string.h>
char *strerror(*error***)**
int *error***;**

strerror helps to generate an error message. It takes the argument *error*, which presumably is an error code generated by an error condition in a program, and may return a pointer to the corresponding error message.

The error numbers recognized and the texts of the corresponding error messages are set by COHERENT.

See Also

perror(), string functions, string.h

Notes

strerror returns a pointer to a static array that may be overwritten by a subsequent call to **strerror**.

strerror differs from the related function **perror** in the following ways: **strerror** receives the error number through its argument *error*, whereas **perror** reads the global constant **errno**. Also, **strerror** returns a pointer to the error message, whereas **perror** writes the message directly into the standard error stream.

The error numbers recognized and the texts of the messages associated with each error number are set by COHERENT. However, **strerror** and **perror** return the same error message when handed the same error number.

string.h — Header File

Declarations for string library
#include <string.h>

string.h is the header that holds the declarations and definitions of all ANSI routines that handle strings and buffers.

See Also

header files

string functions — Overview

The character string is a common formation in C programs. The runtime representation of a string is an array of ASCII characters that is terminated by a null character ('\0'). COHERENT uses this representation when a program contains a string constant; for example:

```
"I am a string constant"
```

The address of the first character in the string is used as the starting point of the string. A pointer to a string holds only this address. Note, too, that an array of 20 characters can hold a string of 19 (*not* 20) non-null characters; the 20th character is the null character that terminates the string.

The following routines are available to help manipulate strings:

index() . . . . . Search string for a character; use **strchr** instead
memchr() . . . . . Search a region of memory for a character
memcmp() . . . . . Compare two regions of memory
memcpy() . . . . . Copy one region of memory into another
memmove() . . . . . Copy one region of memory into another with which it overlaps
memset(). . . . . Fill a region of memory with a character
strcmp() . . . . . Compare two strings
strncmp() . . . . . Compare two lengths for a set number of bytes
strcpy(). . . . . Copy a string
strncmp() . . . . . Copy a portion of a string
strcoll(). . . . . Compare two strings, using locale information
strcspn() . . . . . Return length one string excludes characters in another
strerror(). . . . . Translate an error number into a string
strlen() . . . . . Measure the length of a string
strpbrk() . . . . . Find first occurrence in string of character from another string
strchr(). . . . . Find leftmost occurrence of character in a string
strrchr() . . . . . Find rightmost occurrence of character in a string
strspn(). . . . . Return length one string includes character in another
strstr() . . . . . Find one string within another string
strtok(). . . . . Break a string into tokens
strxfrm() . . . . . Transform a string, using locale information

See their respective entries in the Lexicon for details.

Example

This example reads from **stdin** up to *NNAMES* names, each of which is no more than *MAXLEN* characters long. It then removes duplicate names, sorts the names, and writes the sorted list to the standard output. It demonstrates the functions **shellsort**, **strcat**, **strcmp**, **strcpy**, and **strlen**.

```
#include <stdio.h>

#define NNAMES 512
#define MAXLEN 60

char *array[NNAMES];
char first[MAXLEN], mid[MAXLEN], last[MAXLEN];
char *space = " ";
```

```
int compare();
extern char *strcat();

main()
{
     register int index, count, inflag;
     register char *name;

     count = 0;
     while (scanf("%s %s %s\n", first, mid, last) == 3) {
          strcat(first, space);
          strcat(mid, space);
          name = strcat(first, (strcat(mid, last)));
          inflag = 0;

          for (index=0; index < count; index++)
               if (strcmp(array[index], name) == 0)
                    inflag = 1;

          if (inflag == 0) {
               if ((array[count] =
                    malloc(strlen(name) + 1)) == NULL) {
                    fprintf(stderr, "Insufficient memory\n");
                    exit(1);
               }
               strcpy(array[count], name);
               count++;
          }
     }

     shellsort(array, count, sizeof(char *), compare);
     for (index=0; index < count; index++)
          printf("%s\n", array[index]);
     exit(0);
}

compare(s1, s2)
register char **s1, **s2;
{
     extern int strcmp();
     return(strcmp(*s1, *s2));
}
```

See Also

ASCII, libraries

Notes

The ANSI standard allows adjacent string literals, e.g.:

```
"hello" "world"
```

Adjacent string literals are automatically concatenated. Thus, the compiler will automatically concatenate the above example into:

```
"helloworld"
```

Because this departs from the Kernighan and Ritchie description of C, it will generate a warning message if you use the compiler's **-VSBOOK** option.

strings — Command

Print all character strings from a file
strings [-dopx] [-*length*] [*file* ...]

strings looks for ASCII strings in a binary file. A "string" is defined as any sequence of four or more printable characters. **strings** is useful for identifying unknown object files, or for looking at the messages printed by commands. You can also use it as a filter if *file* is not specified.

strings recognizes the following command-line options:

-d Precede each string by its offset in the file in decimal.

-o Precede each string by its offset in the file in octal.

-p Strip the parity bits of all characters in the string prior to comparison.

-x Precede each string by its offset in the file in hexadecimal.

Finally, the option *-length* forces **strings** to use *length* as the minimum length for a printable string.

See Also

commands, isprint, od

strip — Command

Strip debug, relocation, and symbol tables from executable file
strip -drs *file [...]*

strip removes the symbol table, relocation information, and debug tables from a file. It makes the executable file noticeably smaller.

strip recognizes the following options:

-d Keep debug information.

-r Keep relocation information.

-s Keep symbols.

See Also

cc, commands, ld, nm, size

strlen() — String Function (libc)

Measure the length of a string
#include <string.h>
int strlen(*string***)**
char ****string*;**

strlen measures *string*, and returns its length in bytes, *not* including the null terminator. This is useful in determining how much storage to allocate for a string.

Example

For an example of how to use this function, see the entry for **string**.

See Also

string functions, string.h

strncat() — String Function (libc)

Append one string onto another

#include <string.h>
char *strncat(*string1*, *string2*, *n***)**
char ***string1*, ***string2*; unsigned *n*;

strncat copies up to *n* characters from *string2* onto the end of *string1*. It stops when *n* characters have been copied or it encounters a null character in *string2*, whichever occurs first, and returns the modified *string1*.

Example

For an example of this function, see the entry for **strncpy**.

See Also

strcat(), string functions, string.h

Notes

string1 should point to enough space to hold itself and *n* characters of *string2*. If it does not, a portion of the program or operating system may be overwritten.

strncmp() — String Function (libc)

Compare two strings

#include <string.h>
int strncmp(*string1*, *string2*, *n***)**
char ***string1*, ***string2*; unsigned *n*;

strncmp compares lexicographically the first *n* bytes of *string1* with *string2*. Comparison ends when *n* bytes have been compared, or a null character encountered, whichever occurs first. **strncmp** returns zero if the strings are identical, returns a number less than zero if *string1* occurs earlier alphabetically than *string2*, and returns a number greater than zero if it occurs later. This routine is compatible with the ordering routine needed by **qsort**.

Example

For an example of this function, see the entry for **strncpy**.

See Also

strcmp(), string functions, string.h

strncpy() — String Function (libc)

Copy one string into another

#include <string.h>
char *strncpy(*string1*, *string2*, *n***)**
char ***string1*, ***string2*; unsigned *n*;

strncpy copies up to *n* bytes of *string2* into *string1*, and returns *string1*. Copying ends when *n* bytes have been copied or a null character has been encountered, whichever comes first. If *string2* is less than *n* characters in length, *string2* is padded to length *n* with one or more null bytes.

Example

This example, called **swap.c**, reads a file of names, and changes them from the format

```
first_name  [middle_initial] last_name
```

to the format

```
last_name, first_name [middle_initial]
```

It demonstrates **strncpy**, **strncat**, **strncmp**, and **index**.

```
#include <stdio.h>
#define NNAMES 512
#define MAXLEN 60

char *array[NNAMES];
char gname[MAXLEN], lname[MAXLEN];
extern int strncmp(), strcomp();
extern char *strcpy(), *strncpy(), *strncat(), *index();

main(argc, argv)
int argc; char *argv[];
{
     FILE *fp;
     register int count, num;
     register char  *name, string[60], *cptr, *eptr;
     unsigned glength, length;

     if (--argc != 1) {
          fprintf (stderr, "Usage: swap filename\n");
          exit(1);
     }

     if ((fp = fopen(argv[1], "r")) == NULL)
          printf("Cannot open %s\n", argv[1]);
     count = 0;

     while (fgets(string, 60, fp) != NULL) {
          if ((cptr = index(string, '.')) != NULL) {
               cptr++;
               cptr++;
          } else if ((cptr = index(string,' ')) != NULL)
               cptr++;

          strcpy(lname, cptr);
          eptr = index(lname, '\n');
          *eptr = ',';

          strcat(lname," ");
          glength = (unsigned)(strlen(string) - strlen(cptr));
          strncpy(gname, string, glength);

          name = strncat(lname, gname, glength);
          length = (unsigned)strlen(name);
          array[count] = malloc(length + 1);
```

```
            strcpy(array[count],name);
            count++;
        }

        for (num = 0; num < count; num++)
            printf("%s\n", array[num]);
        exit(0);
}
```

See Also

strcpy(), string functions, string.h

Notes

string1 must point to enough space to *n* bytes; otherwise, a portion of the program or operating system may be overwritten.

strpbrk() — String Function (libc)

Find first occurrence of a character from another string
#include <string.h>
char *strpbrk(*string1*, *string2***)**
char ***string1*, ***string2*;

strpbrk returns a pointer to the first character in *string1* that matches any character in *string2*. It returns NULL if no character in *string1* matches a character in *string2*.

The set of characters that *string2* points to is sometimes called the "break string". For example,

```
char *string = "To be, or not to be: that is the question.";
char *brkset = ",;";
strpbrk(string, brkset);
```

returns the value of the pointer **string** plus five. This points to the comma, which is the first character in the area pointed to by **string** that matches any character in the string pointed to by **brkset**.

See Also

string functions, string.h

Notes

strpbrk resembles the function **strtok** in functionality, but unlike **strtok**, it preserves the contents of the strings being compared. It also resembles the function **strchr**, but lets you search for any one of a group of characters, rather than for one character alone.

strrchr() — String Function (libc)

Search for rightmost occurrence of a character in a string
#include <string.h>
char *strrchr(*string*, *character***)**
char ***string*; **int *character*;**

strrchr looks for the last, or rightmost, occurrence of *character* within *string*. *character* is declared to be an **int**, but is handled within the function as a **char**. Another way to describe this function is to say that it performs a reverse search for a character in a string. It is equivalent to the COHERENT function **rindex**.

strrchr returns a pointer to the rightmost occurrence of *character*, or NULL if *character*

could not be found within *string*.

See Also

rindex(), string functions, string.h

strspn() — String Function (libc)

Return length a string includes characters in another
#include <string.h>
unsigned int strspn(*string1*, *string2***)**
char ****string1*; char ****string2***;**

strspn returns the length for which *string1* initially consists only of characters that are found in *string2*. For example,

```
char *s1 = "hello, world";
char *s2 = "kernighan & ritchie";
strcspn(s1, s2);
```

returns two, which is the length for which the first string initially consists of characters found in the second.

See Also

string functions, string.h

strstr() — String Function (libc)

Find one string within another
#include <string.h>
char *strstr(*string1*, *string2***)**
char ****string1*, ****string2***;**

strstr looks for *string2* within *string1*. The terminating null character is not considered part of *string2*.

strstr returns a pointer to where *string2* begins within *string1*, or NULL if *string2* does not occur within *string1*.

For example,

```
char *string1 = "Hello, world";
char *string2 = "world";
strstr(string1, string2);
```

returns **string1** plus seven, which points to the beginning of **world** within **Hello, world**. On the other hand,

```
char *string1 = "Hello, world";
char *string2 = "worlds";
strstr(string1, string2);
```

returns NULL because **worlds** does not occur within **Hello, world**.

See Also

string functions, string.h

strtod() — General Function (libc)

Convert string to floating-point number
#include <stdlib.h>
double strtod(*string, tailptr***)**
char ****string*; char *****tailptr***;**

strtod converts the number given in *string* to a double-precision floating-point number and returns its value. It is a more general version of the function **atof**. **strtod** also stores a pointer to the first character following the number through *tailptr*, provided *tailptr* is not **NULL**.

strtod parses the input *string* into three portions: beginning, subject sequence, and tail.

The *beginning* consists of zero or more white-space characters that begin the string.

The *subject sequence* is the portion of the input *string* that **strtod** converts into a floating-point number. It consists of an optional sign character, a nonempty sequence of decimal digits optionally including a decimal-point character, and an optional exponent. If present, the exponent consists of either 'e' or 'E' followed by an optional sign and a nonempty sequence of decimal digits. **strtod** reads characters until it encounters either a second decimal-point character or exponent marker, or any other non-numeral.

The *tail* continues from the end of the subject sequence to the null character that ends the string.

strtod ignores the beginning portion of the string. It converts the subject sequence to a double-precision number. Finally, it sets the pointer pointed to by *tailptr* to the address of the first character of the string's tail.

strtod returns the **double** generated from the subject sequence. If no subject sequence could be recognized, it returns zero and stores the initial value of *string* through *tailptr*. If the number represented by the subject sequence is too large or too small to fit into a **double**, then **strtod** sets the global constant **errno** to **ERANGE** and returns **HUGE_VAL** or zero, respectively.

See Also

atof, double, errno, general functions, limits.h, stdlib.h, strtol, strtoul

Notes

strtod ignores initial white space in the string pointed to by *string*; white space is defined as being all characters so recognized by the function **isspace**.

strtok() — String Function (libc)

Break a string into tokens
#include <string.h>
char *strtok(*string1, string2***);**
char ****string1*, ****string2***;**

strtok helps to divide a string into a set of tokens. *string1* points to the string to be divided, and *string2* points to the character or characters that delimit the tokens.

strtok divides a string into tokens by being called repeatedly.

On the first call to **strtok**, *string1* should point to the string being divided. **strtok** searches for a character that is *not* included within *string2*. If it finds one, then **strtok** regards it as the beginning of the first token within the string. If one cannot be found, then **strtok** returns NULL to signal that the string could not be divided into tokens. When the beginning of the first token is found, **strtok** then looks for a character that *is* included

within *string2*. When one is found, **strtok** replaces it with a null character to mark the end of the first token, stores a pointer to the remainder of *string1* within a static buffer, and returns the address of the beginning of the first token.

On subsequent calls to **strtok**, set *string1* to NULL. **strtok** then looks for subsequent tokens, using the address that it saved from the first call. With each call to **strtok**, *string2* may point to a different delimiter or set of delimiters.

See Also

string functions, string.h

strtok() — String Function (libc)

Break a string into tokens

#include <string.h>
char *strtok(*string1*, *string2*)
**char ***string1*, *****string2*;

strtok helps to divide a string into a set of tokens. *string1* points to the string to be divided, and *string2* points to the character or characters that delimit the tokens.

strtok divides a string into tokens by being called repeatedly.

On the first call to **strtok**, *string1* should point to the string being divided. **strtok** searches for a character that is *not* included within *string2*. If it finds one, then **strtok** regards it as the beginning of the first token within the string. If one cannot be found, then **strtok** returns NULL to signal that the string could not be divided into tokens. When the beginning of the first token is found, **strtok** then looks for a character that *is* included within *string2*. When one is found, **strtok** replaces it with a null character to mark the end of the first token, stores a pointer to the remainder of *string1* within a static buffer, and returns the address of the beginning of the first token.

On subsequent calls to **strtok**, set *string1* to NULL. **strtok** then looks for subsequent tokens, using the address that it saved from the first call. With each call to **strtok**, *string2* may point to a different delimiter or set of delimiters.

Example

The following example breaks **command_string** into individual tokens and puts pointers to the tokens into the array **tokenlist**[]. It then returns the number of tokens created. No more than **maxtoken** tokens will be created. **command_string** is modified to place '\0' over token separators. The token list points into **command_string**. Tokens are separated by spaces, tabs, commas, semicolons, and newlines.

```
#include <stdlib.h>
#include <string.h>
#include <stddef.h>
#include <stdio.h>

tokenize(command_string, tokenlist[], maxtoken)
char *command_string, *tokenlist[]; size_t maxtoken;
{
     static char tokensep[]="\t\n ,;";
     int tokencount;
     char *thistoken;

     if(command_string == NULL || !maxtoken)
          return 0;
```

```
        thistoken = strtok(command_string, tokensep);

        for(tokencount = 0; tokencount < maxtoken &&
                    thistoken != NULL;) {
             tokenlist[tokencount++] = thistoken;
             thistoken = strtok(NULL, tokensep);
        }

        tokenlist[tokencount] = NULL;
        return tokencount;
}

#define MAXTOKEN 100
char *tokens[MAXTOKEN];
char buf[80];

main()
{
        for(;;) {
             int i, j;

             printf("Enter string ");
             fflush(stdout);
             if(gets(buf) == NULL)
                  exit(0);

             i = tokenize(buf, tokens, MAXTOKEN);
             for (j = 0; j < i; j++)
                  printf("%s\n", tokens[j]);
        }
}
```

See Also

string functions, string.h

strtol() — General Function (libc)

Convert string to long integer

#include <stdlib.h>

long strtol(*string, tailptr, base*)

char ***string*; **char** ****tailptr*; **int** *base*;

strtol converts the number given in *string* to a **long** and returns its value; it is a more general version of the function **atol**. **strtol** also stores a pointer to the first character following the number through *tailptr*, provided *tailptr* is not **NULL**.

base gives the base of the number being read, either 0 or a value from 2 to 36. If the given *base* is zero, **strtol** determines an implicit base for the number: hexadecimal if the number starts with **0x** or **0X**, octal if the number starts with **0**, or decimal otherwise. Alternatively, you can specify a *base* between 2 and 36.

strtol parses *string* into three portions: beginning, subject sequence, and tail.

The *beginning* consists of zero or more white-space characters that begin the string.

The *subject sequence* is the portion of the string that **strtol** converts into a **long**. It consists of an optional sign character, an optional prefix **0x** or **0X** if the *base* is 16, and a nonempty

sequence of *digits* in the specified base. For example, if the *base* is 16, then **strtol** recognizes numeric characters '0' to '9' and alphabetic characters 'A' through 'F' and 'a' to 'f' as digits. It continues to scan until it encounters a nondigit.

The *tail* continues from the end of the subject sequence to the null character that ends the string.

strtol ignores the beginning portion of the string. It converts the subject sequence to a **long**. Finally, if *tailptr* is not **NULL**, it sets the pointer pointed to by *tailptr* to the address of the first character of the string's tail.

strtol returns a **long** representing the value of the subject sequence. If the input *string* does not specify a valid number, it returns zero and stores the initial value of *string* through *tailptr*. If the number it builds is too large or too small to fit into a **long**, it sets the global variable **errno** to the value of the macro **ERANGE** and returns **LONG_MAX** or **LONG_MIN**, respectively.

See Also

atol, errno, general functions, limits.h, long, stdlib.h, strtoul

Notes

strtol ignores initial white space in the input *string*. White space is defined as being all characters so recognized by the function **isspace**.

strtoul() — General Function (libc)

Convert string to unsigned long integer
#include <stdlib.h>
unsigned long strtoul(*string, tailptr, base*)
char **string*; **char** ***tailptr*; **int** *base*;

strtoul converts the number given in *string* to a **unsigned long** and returns its value. It is the **unsigned long** counterpart of **strtol** and a more general version of the function **atol**. **strtoul** also stores a pointer to the first character following the number through *tailptr*, provided *tailptr* is not **NULL**.

base gives the base of the number being read, either 0 or a value from 2 to 36. If the given *base* is zero, **strtoul** determines an implicit base for the number: hexadecimal if the number starts with **0x** or **0X**, octal if the number starts with **0**, or decimal otherwise. Alternatively, the user can specify an explicit *base* between 2 and 36.

strtoul parses the *string* into three portions: beginning, subject sequence, and tail.

The *beginning* consists of zero or more white-space characters that begin the string.

The *subject sequence* is the portion of the string that **strtoul** converts into an **unsigned long**. It consists of an optional sign character, an optional prefix **0x** or **0X** if the *base* is 16, and a nonempty sequence of *digits* in the specified base. For example, if the *base* is 16, then **strtoul** recognizes numeric characters '0' to '9' and alphabetic characters 'A' through 'F' and 'a' to 'f' as digits. It continues to scan until it encounters a nondigit.

The *tail* continues from the end of the subject sequence to the null character that ends the string.

strtoul ignores the beginning portion of the string. It converts the subject sequence to an **unsigned long**. Finally, if *tailptr* is not **NULL**, it sets the pointer pointed to by *tailptr* to the address of the first character of the string's tail.

strtoul returns an **unsigned long** representing the value of the subject sequence. If the input *string* does not specify a valid number, it returns zero and stores the initial value of

string through *tailptr*. If the number it builds is too large to fit into an **unsigned long**, it sets the global variable **errno** to the value of the macro **ERANGE** and returns **ULONG_MAX**.

Example

This example uses **strtoul** as a hash function for table lookup. It demonstrates both hashing and linked lists. Hash-table lookup is the most efficient when used to look up entries in large tables; this is an example only.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*
 * For fastest results, use a prime about 15% bigger
 * than the table. If short of space, use a smaller prime.
 */
#define HASHP 11
struct symbol {
     struct symbol *next;
     char *name;
     char *descr;
} *hasht[HASHP], codes[] = {

     NULL,      "a286",          "frogs togs",
     NULL,      "xy7800",        "doughnut holes",
     NULL,      "z678abc",       "used bits",
     NULL,      "xj781",         "black-hole varnish",
     NULL,      "h778a",         "table hash",
     NULL,      "q167",          "log(-5.2)",
     NULL,      "18888",         "quid pro quo",
     NULL,      NULL,            NULL   /* end marker */
};

void
buildTable(void)
{
     long h;
     register struct symbol *sym, **symp;

     for(symp = hasht; symp != (hasht + HASHP); symp++)
          *symp = NULL;

     for(sym = codes; sym->descr != NULL; sym++) {
          /*
           * hash by converting to base 36. There are
           * many ways to hash, but use all the data.
           */
```

```
            h = strtoul(sym->name, NULL, 36) % HASHP;
            sym->next = hasht[h];
            hasht[h] = sym;
        }
}

struct symbol *
lookup(char *s)
{
        long h;
        register struct symbol *sym;

        h = strtoul(s, NULL, 36) % HASHP;
        for(sym = hasht[h]; sym != NULL; sym = sym->next)
            if(!strcmp(sym->name, s))
                return(sym);
        return(NULL);
}

main(void)
{
        char buf[80];
        struct symbol *sym;

        buildTable();
        for(;;) {
            printf("Enter name ");
            fflush(stdout);

            if(gets(buf) == NULL)
                exit(EXIT_SUCCESS);

            if((sym = lookup(buf)) == NULL)
                printf("%s not found\n", buf);

            else
                printf("%s is %s\n", buf, sym->descr);
        }
}
```

See Also

errno, general functions, limits.h, stdlib.h, strtol

Notes

strtoul ignores initial white space in the input *string*. White space is defined as being all characters so recognized by the function **isspace**.

struct — C Keyword

Data type

struct is a C keyword that introduces a structure. The following is an example of how **struct** can be used in the description of a name and address file:

```
struct address {
     char firstname[10];
     char lastname[15];
     char street[25];
     char city[10];
     char state[2];
     char zip[5];
     int  salescode;
};
```

The C Programming Language prohibits the assignment of structures, the passing of structures to functions, and the returning of structures by functions. COHERENT, however, lifts these restrictions. It allows one structure to be assigned to another, provided the two structures are of the same type. It also allows structures to be passed by and returned by functions. These features are supported by most compilers, but users should be aware that their use can cause problems in porting code to some compilers.

See Also

array, C keywords, field, initialization, structure

structure — Definition

A **structure** is a set of variables that has been given a name and can be manipulated as a single entity. The variables may be of different data types. Structures are a convenient way to deal with data elements that belong together, such as names and addresses, employee descriptions, or sales and inventory information.

See Also

definitions, struct

structure assignment — Technical Information

The C Programming Language forbids structure assignment, the passing of structures to functions, and returning structures from functions (as opposed to the passing or returning of *pointers* to structures). The COHERENT C compiler lifts these restrictions.

Some C compilers transform structure arguments and structure returns into structure pointers. Note that the use of structure assignment, structure arguments, or structure returns may create problems when porting the code to another C compiler.

See Also

portability, struct, structure, technical information

Notes

Because this feature deviates from the description of the C language found in the first edition of *The C Programming Language*, compiling with the **-VSBOOK** option will flag all points where it occurs in your program.

strxfrm() — String Function (libc)

Transform a string
#include <string.h>
unsigned int strxfrm(*string1*, *string2*, *n*)
char *****string1*, *****string2*; **unsigned int** *n*);

strxfrm transforms *string2* using information concerning the program's locale, as set by the function **setlocale**.

strxfrm writes up to *n* bytes of the transformed result into the area pointed to by *string1*. It returns the length of the transformed string, not including the terminating null character. The transformation incorporates locale-specific material into *string2*.

If *n* is set to zero, **strxfrm** returns the length of the transformed string.

If two strings return a given result when compared by **strcoll** before transformation, they will return the same result when compared by **strcmp** after transformation.

See Also

string functions, string.h

Notes

If **strxfrm** returns a value equal to or greater than *n*, the contents of the area pointed to by *string1* are indeterminate.

COHERENT has not yet implemented the ANSI locale functions. Therefore, **strxfrm** behaves the same as **strcpy**.

stty() — COHERENT System Call (libc)

Set terminal modes
#include <sgtty.h>
int stty(*fd, sgp***)**
int *fd***;**
struct sgttyb ***_sgp_;**

The COHERENT system call **stty** sets a terminal's attributes. See the Lexicon article for **stty** for information on terminal attributes and their legal values.

Example

This example demonstrates both **stty** and **gtty**. It sets terminal input to read one character at a time (that is, it reads the terminal in "raw" form). When you type 'q', it restores the terminal to its previous settings, and exits. For an additional example, see the **pipe** Lexicon article.

```
#include <sgtty.h>

main()
{
        struct sgttyb os, ns;
        char buff;

        printf("Waiting for q\n");
        gtty(1, &os);           /* save old state */
        ns = os;                /* get base of new state */
        ns.sg_flags |= RAW;     /* prevent <ctl-c> from working */
        ns.sg_flags &= ~(ECHO|CRMOD);/* no echo for now... */
        stty(1, &ns);           /* set mode */

        do {
                buff = getchar();   /* wait for the keyboard */
        } while(buff != 'q');

        stty(1, &os);       /* reset mode */
}
```

Files

<sgtty.h> — Header file

See Also

COHERENT system calls, exec, gtty(), ioctl(), open(), read(), sgtty.h, stty, write()

Notes

Please note that if you use **stty** to change the baud rate on a port, you must first invoke **sleep()**. If you do not, the port reverts back to its default settings.

stty — Command

Set/print terminal modes
stty [*option* ...]

If no *option* is specified, **stty** prints the modes of the standard output device in the standard error stream. Otherwise, each *option* modifies the modes of the standard output device. The device is usually a terminal, although tapes, disks and other special files may be applicable.

In normal processing ("cooked" mode), the *erase* and *kill* characters (normally **<crtl-H>** and **<ctrl-U>**) erase, respectively, one typed character and a typed line. The *stop-output* and *start-output* characters (normally **<ctrl-S>** and **<ctrl-Q>**) stop and restart output. The *interrupt* character (normally **DELETE** or **RUBOUT** ASCII 0177), sends the signal **SIGINT**, which usually terminates program execution. The *quit* character (normally ASCII 034, **FS**, which differs on various terminals but is often **<ctrl-\>**) sends the signal **SIGQUIT**, which usually terminates program execution with a core dump. The *end of file* character (normally **<ctrl-D>**) generates an end of file from the terminal. Each special character can be changed with the appropriate *option*.

On some machines, the default characters differ from those given above. On the IBM Personal Computer, for example, the default kill character is **<ctrl-U>** and the default interrupt character is **<ctrl-C>**.

The following table describes each available *option*. The *c* argument may be a literal character or may be of the form '^X' for **<ctrl-X>**.

number	Set input and output baud rates of the device to the speed *number*, if possible.
0	Hang up phone immediately.
-a	Display all modes.
break *c*	Set the break character to *c*.
cbreak	Break after every input character. This allows a program to return after having read *N* characters from a terminal, even if no end of file, break or newline character was typed.
-cbreak	Exit from cbreak mode.
cooked	Exit from raw mode.
crt	Terminal is a CRT. Echoing is enhanced.
-crt	The terminal is not a CRT.
echo	Echo characters as they are received on the input.

-echo Disable echoing.

ek Set the erase character to '#' and the kill character to '@'.

eof *c* Set the end of file character to *c*.

erase *c* Set the erase character to *c*.

even Accept even-parity characters.

-even Do not accept even-parity characters.

excl Exclusive use: subsequent opens will fail.

-excl Non-exclusive use.

flush Flush characters waiting in output or input queues.

-flush Do not flush characters.

hup Hang up the phone on last close.

-hup Do not hang up on last close.

int *c* Set the interrupt character to *c*.

kill *c* Set the kill character to *c*.

nl Disable newline mapping.

-nl Enable newline mapping: map carriage returns to linefeeds on input, and append carriage returns before linefeeds on output.

odd Accept odd-parity characters.

-odd Do not accept odd-parity characters.

print Print terminal attributes.

quit *c* Set the quit character to *c*.

raw Raw mode: suppress all processing and mapping (except echo).

-raw Exit from raw mode.

rawin Suppress all processing and mapping on the input stream.

-rawin Exit from rawin mode.

rawout Suppress all processing and mapping on the output stream.

-rawout Exit from rawout mode.

sane Set the terminal to a known state.

start *c* Set the start-output character to *c*.

stop *c* Set the stop-output character to *c*.

tabs Do not expand tabs: useful for terminals which process tabs internally.

-tabs Expand tabs to the appropriate number of spaces on output. The system assumes tabstops are at every eighth column.

tandem Tandem mode. The system will send the programmed stop-output character whenever there is a danger of losing characters from the input stream due to buffering limitations. The system will send the start-character when the level of

unprocessed characters has subsided.

-tandem Disable tandem mode.

See Also

ASCII, commands, getty, init, ioctl(), signal()

Notes

The system does not support character delays or mapping upper to lower case.

su — Command

Substitute user id, become superuser
su [*user* [*command*]]

Default *user* is **root**; default *command* is **sh**. **su** changes the real user id and the effective user id to that of the *user*. If *user* has a login password, **su** requests it. Then it executes the specified *command*.

If *command* is absent, **su** invokes an interactive sub-shell.

If *user* is absent, **su** assumes user name **root** (the superuser).

Files

/etc/passwd — Login names and passwords

See Also

commands, login, newgrp, sh, superuser

suload() — COHERENT System Call

Unload device driver
#include <con.h>
int suload(*major*)
int *major*;

The COHERENT system accesses all devices through drivers residing in the system. Except for the root device, drivers must be explicitly loaded before use; this operation does not involve re-booting.

suload unloads the driver identified by *major*, which was previously loaded by a call to **sload**. This call is restricted to the superuser.

Files

<con.h>
/drv/*

See Also

init, l.out.h, ld, load, sload

Diagnostics

suload returns zero upon successful unloading of the appropriate driver, or -1 on errors. It fails if the driver *major* is not loaded.

sum — Command

Print checksum of a file
sum [*file* ...]

sum prints an unsigned integer checksum and a size in blocks (rounding up) for each *file* specified. If more than one *file* is specified, **sum** also prints the file name. If no *file* is

specified, **sum** reads the standard input.

sum may be used to verify the integrity of data transferred across phone lines or stored on an unreliable medium.

See Also

cmp, commands

superuser — Definition

The *superuser* is the user who has system-wide permissions. He can execute any program, read any file, and write into any directory. Thus, superuser status is reserved to the system administrator, also called **root**, who needs this status to control the operation of the system.

No person should be able to become the superuser without knowing a password. Because the superuser in effect "owns" the system, the superuser password should be guarded most carefully.

See Also

definitions, root, su

swab() — General Function (libc)

Swap a pair of bytes

void swab(*src*, *dest*, *nb*) **char** **src*, **dest*; **unsigned** *nb*;

The ordering of bytes within a word differs from machine to machine. This may cause problems when moving binary data between machines. **swab** interchanges each pair of bytes in the array *src* that is *n* bytes long, and places the result into the array *dest*. The length *nb* should be an even number, or the last byte will not be touched. *src* and *dest* may be the same place.

Example

This example prompts for an integer; it then prints the integer both as you entered it, and as it appears with its bytes swapped.

```
#include <stdio.h>

main()
{
     int word;

     printf("Enter an integer: \n");
     scanf("%d", &word);
     printf("The word is 0x%x\n", word);
     swab(&word, &word, 2);
     printf("The word with bytes swapped is 0x%x\n", word);
}
```

See Also

dd, canon.h, general functions

switch — C Keyword

Test a variable against a table

switch is a C keyword that lets you perform a number of tests on a variable in a convenient manner. For example,

```
while(foo < 10)
      switch(foo) {
      case 1:
            dosomething();
            break;
      case 2:
            somethingelse();
      case 3:
            anotherthing();
            break;
      default:
            break;
      }
}
```

is equivalent to

```
while(foo < 10) {
      if(foo == 1) {
            dosomething();
            continue;
      } else if (foo == 2) {
            somethingelse();
            anotherthing();
            continue;
      } else if(foo == 3) {
       /* Note: compiler eliminates duplicate code */
            anotherthing();
            continue;
      } else
            break;
}
```

switch is always used with the **case** statement, and nearly always with the **default** statement.

See Also

break, C keywords, case, default, keyword, while

sync() — COHERENT System Call (libc)

Flush system buffers

sync()

sync() is the COHERENT system call that copies the contents of all memory buffers to disk.

See Also

COHERENT system calls

sync — Command

Flush system buffers
sync

Most COHERENT commands manipulate files stored on a disk. To improve system performance, the COHERENT system often changes a copy of part of the disk in a buffer in memory, rather than repeatedly performing the time-consuming disk access required.

sync writes information from the memory buffers to the disk, updating the disk images of all mounted file systems which have been changed. In addition, it writes the date and time on the root file system.

sync should be executed before system shutdown to ensure the integrity of the file system.

See Also

commands

system() — General Function (libc)

Pass a command to the shell for execution
int system(*commandline***) char ****commandline***;**

system passes *commandline* to the shell **sh**, which loads it into memory and executes it. **system** executes commands exactly as if they had been typed at the COHERENT command level. **system** may be used by commands such as **ed**, which can pass commands to the COHERENT shell in addition to processing normal interactive requests.

Example

This example uses **system** to list the names of all C source files in the parent directory.

```
#include <stdio.h>
main()
{
	system("cd .. ; ls *.c > mytemp; cat mytemp");
}
```

See Also

exec, fork(), general functions, popen(), wait

Diagnostics

system returns the exit status of the child process, in the format described in **wait**: exit status in the high byte, signal information in the low byte. Zero normally means success, whereas nonzero normally means failure. This, however, depends on the *command*. If the shell is not executable, **system** returns a special code of octal 0177.

system maintenance — Overview

The COHERENT system automatically invokes a number of utilities that help COHERENT to maintain itself. These utilities will, for example, run programs for you at pre-determined times, swap temporary files in and out of memory, update files, and perform other useful tasks. COHERENT includes the following system maintenance routines:

aliases . . . . . File of users' aliases
atrun . . . . . Execute programs at a preset time
boottime. . . . . Time of last system boot
brc . . . . . Perform maintenance chores, single-user mode

checklist . . . . . File systems to check when booting COHERENT
cron . . . . . Execute commands periodically
getty . . . . . Terminal initialization
hpd . . . . . Spooler daemon for Hewlett-Packard LaserJet printer
init . . . . . System initialization
logmsg . . . . . File that holds login prompt
lpd . . . . . Line printer spooler daemon
modemcap . . . . . Modem-description language
modeminit . . . . . Initialize a modem
motd . . . . . File that holds message of the day
mount.all . . . . . File systems to mount when booting COHERENT
paths . . . . . Routing data base for mail
profile . . . . . Set user's environment at login
rc . . . . . Perform standard maintenance chores
termcap . . . . . Terminal-description language
update . . . . . Update file systems periodically
uucpname . . . . . Set system's UUCP name

See Also

Lexicon

T

tail — Command

Print the end of a file
tail [+*n*[**bcfl**]] [*file*]
tail [-*n*[**bcfl**]] [*file*]

tail copies the last part of *file*, or of the standard input if none is named, to the standard output.

The given *number* tells **tail** where to begin to copy the data. Numbers of the form *+number* measure the starting point from the beginning of the file; those of the form *-number* measure from the end of the file.

A specifier of blocks, characters, or lines (*b*, **c**, or **l**, respectively) may follow the number; the default is lines. If no *number* is specified, a default of -10 is assumed.

The **-f** option opens the tail of a file, and then displays new material as it is added to a file. This command lets you watch a file as it is being built, such as by **nroff**. Note that when **tail** is invoked with this option, it does not exit; therefore, when you wish to exit, type the interrupt character.

See Also

commands, dd, egrep, head, sed

Notes

Because **tail** buffers data measured from the end of the file, large counts may not work.

tan() — Mathematics Function (libm)

Calculate tangent
#include <math.h>
double tan(*radian*) **double** *radian*;

tan calculates the tangent of its argument *radian*, which must be in radian measure.

Example

For an example of this function, see the entry for **acos**.

See Also

mathematics library, tanh()

Diagnostics

tan returns a very large number where it is singular, and sets **errno** to **ERANGE**.

tanh() — Mathematics Function (libm)

Calculate hyperbolic cosine
#include <math.h>
double tanh(*radian*) **double** *radian*;

tanh calculates the hyperbolic tangent of *radian*, which is in radian measure.

Example

For an example of this function, see the entry for **cosh**.

See Also

mathematics library

Diagnostics

tanh sets **errno** to **ERANGE** when an overflow occurs.

tape — Device Driver

Magnetic tape devices

This section gives a general explanation of COHERENT's use of industry-standard half-inch, nine-track magnetic tape. Exceptions or additional information may be found in sections of this manual describing particular devices.

A tape volume contains files, each consisting of one or more records and terminated by a tape mark. Two tape marks terminate the last file. Tape records may vary in length, but cannot exceed 2^16 bytes (2^15 is more practical).

Like other block-oriented devices, tape units may be accessed through the system's *cooked* interface or through the *raw* interface. On a cooked device, seeking to any byte offset and reading in any number of bytes is possible. It is not possible to read beyond the tape mark at the end of the current file. All records in the file must be 512 bytes long, except the last. Write requests must be made in increments of 512 bytes, except the last. A cooked tape may be mounted like a disk, but only as a read-only file system.

A raw device bypasses the buffer cache, so I/O occurs directly to or from the user's buffer. One write request generates one tape record, and one read request returns exactly one record. The number of bytes read may be less than expected. If the tape mark is read, a count of zero is returned, but the system positions the tape at the start of the next tape file. Seeking on a raw device is ignored, and mounting is not allowed.

A unit cannot be opened if it is off-line or already in use. If the write ring is absent, the unit cannot be opened for writing. Closing the device has varying effects, depending on the minor device opened and whether the device was opened for reading or writing. In the case of reading, the tape is rewound; if the no-rewind option was specified, the tape advances to the next file. In the case of writing, two tape marks are written at the current position and the tape is rewound; if the no-rewind option was specified, two tape marks are written and the tape is positioned between them. When you close a device that had been opened for writing, the tape volume ends at the current position; data beyond this point are undefined.

The following device options exist, selected by prefixes to the device name:

h Read or write data at high density. The exact density depends on the drive model, but 1600 BPI (high) and 800 BPI (low) are typical.

n Do not rewind on close.

r The device is raw.

Hard errors may occur during tape operation. They include detection of the end-of-tape (EOT) reflector, reading an unexpectedly long record, or seeking a cooked tape into a tape mark. After an error, no further operations may be performed on the unit until the program closes the device and the operator rewinds the tape. Soft parity errors may arise due to dirt, bad tape or misaligned heads. On writes, the driver attempts to place the record further along the tape. On reads, the driver simply rescans the record. After several failures, the driver announces a hard error.

Most utilities use generic device names, which are links to the actual device files appropriate for the site.

Files

/dev/mt — Generic cooked tape device
/dev/rmt — Generic raw tape device

See Also

device drivers

Diagnostics

Drivers may report errors to the console.

Notes

Not every edition of COHERENT supports magnetic tape.

tar — Command

V7 tape archive manager
tar [crtux[0-7bflmvwU] [*blocks*] [*archive*] *file* ...

tar is a utility that lets you read, write, and update archives in a machine-independent format. Its name is an abbreviation for *tape archive;* however, **tar** is a flexible utility that can read/write output to files and floppy disks, as well as to magnetic tape.

The first argument of the command line must contain exactly one directive character, followed by zero or more option characters. *file* is the file to be written into or extracted from the archive. If *file* is a directory, **tar** processes its contents recursively. For directives that read an archive, the absence of a *file* argument tells **tar** to process every file in the archive. For directives that write to an archive, the absence of a *file* argument tells **tar** to process every file in the current directory.

The directives are as follows:

c Create a new archive. Overwrite the previous contents of the archive.

r Replace (append) the named files in the archive.

t Write a table of contents of the archive to the standard output.

u Update the archive by replacing the named files that are newer (*mtime* larger) than any version in the archive.

x Extract the named files from the archive. Overwrite identically names files. **tar** extracts each version of each file, leaving the latest version at the end.

The options are as follows:

0-7 A single octal digit specifies a tape drive on which the archive may be found. **tar** concatenates this digit to the default tape name **/dev/mt** to form the path name accessed. This option, of course, is available only to COHERENT systems that support a nine-track magnetic tape drive.

b The next argument is a number between one and 20, specifying how many **blocks** are to be written in each archive. **tar** determines the blocking factor automatically on input. When the blocking factor is not one, **tar** automatically writes its output to device **/dev/rmt**, i.e., the raw tape-drive device.

f The next argument names the **archive**. If the argument is a hyphen '-', it signifies the standard input for input directives and the standard output for output directives.

l **tar** preserves links within the structure it writes into its archive, but breaks any links across the boundary of the structure. This option requests that **tar** report all such broken links.

m Ignore the *mtime* for each extracted file. By default, **tar** restores the *mtime* for each extracted file.

v Verbose flag. If directive is **t**, the output for each file includes its mode, group id, user id, size, and mtime, in addition to its path name. Otherwise, **tar** writes the directive and the path name to the standard output for input directives or the standard error for output directives as each file is processed.

w For each file to be processed, **tar** writes the directive and path name to the terminal device, then reads a line from that device and acts on it as follows:

- **n** Skip the file.
- **y** Process the file.
- **x** Exit immediately.

An empty response is treated as **n**, and end of file is treated as **x**. If a directory is skipped, all its contents are skipped. If included, all its contents are processed with this option.

U The version of **tar** found on some UNIX systems have following bug: when the blocking factor is not one, the last few blocks of the last record written may be garbage. This bug is described elsewhere by other symptoms. This option says that the tape was created by the buggy program, so the trailing garbage should be ignored.

Examples

To **tar** the contents of directory **piggy** into file **piggy.tar**, use the command:

```
tar cf piggy piggy.tar
```

disk formatted with fdformat -av /dev/rfva0

To **tar** files to a floppy disk, it is sufficient to have a floppy disk formatted with the command **fdformat**. The floppy does not have to have a COHERENT file system on it. For example, to **tar** directory **stephen** to a high-density, 5.25-inch, formatted floppy disk in drive 0, use the following command:

```
tar cf /dev/fha0 stephen
```

Note that this permits you to **tar** only one archive per floppy disk. To read files from this archive, use the command:

```
tar xf /dev/fha0
```

Files

/dev/mt* — Default tape device
/dev/rmt* — Default tape device for blocking factor greater than one

See Also

commands, cpio, dump, link(), restor, stat(), ustar

Notes

Path names must be less than 100 characters. The **m** option does not affect directories. The only way to extract the *N*th version of a file is with the **w** option.

technical information — Overview

The Lexicon includes the following entries that describe technical aspects of COHERENT:

aliases . . . . . File of users' aliases
ASCII . . . . . ASCII table
booting. . . . . How booting works
byte ordering . . . . . Machine-dependent order of bytes
calling conventions
data formats
data types
environ. . . . . Process environment
errno . . . . . External integer for return of error status
execution . . . . . Program execution
floppy disks . . . . . Information about floppy disks
hard disk . . . . . Information about hard disks
keyboard tables . . . . . How to write a keyboard table
man. . . . . Manual macro package
memory allocation
misc . . . . . Archive of miscellaneous library functions
modem. . . . . Information about modems
modem control. . . . . Information about controlling modems
ms. . . . . Manuscript macro package
MS-DOS . . . . . That other operating system
portability
printer . . . . . Information about printers
rename. . . . . How to rename a file
security
signame . . . . . Array of names of signals
storage class
structure assignment
terminal . . . . . Information about terminals
type checking
type promotion

See Also

Lexicon

tee — Command

Branch pipe output
tee [**-a**] [**-i**] [*file* ...]

tee reads from standard input, usually a pipe, and writes to the standard output, usually a pipe. **tee** also writes a copy of the input data to each *file* specified.

The **-a** flag tells **tee** to append data to each *file*, analogous to the shell construct ">>*file*". Otherwise, it creates each *file*, analogous to the construct ">*file*".

The flag **-i** means ignore interrupts.

See Also

commands, ksh, sh

tempnam() — General Function (libc)

Generate a unique name for a temporary file
char *tempnam(*directory, name***);**
char ****directory*, ****name***;**

tempnam constructs a unique temporary name that can be used to name a file.

directory points to the name of the directory in which you want the temporary file written. If this variable is NULL, **tempnam** reads the environmental variable **TMPDIR** and uses it for *directory*. If neither *directory* nor **TMPDIR** is given, **tempnam** uses **/tmp**.

name points to the string of letters that will prefix the temporary name. This string should not be more than three or four characters, to prevent truncation or duplication of temporary file names. If *name* is NULL, **tempnam** sets it to **t**.

tempnam uses **malloc** to allocate a buffer for the temporary file name it returns. If all goes well, it returns a pointer to the temporary name it has written. Otherwise, it returns NULL if the allocation fails or if it cannot build a temporary file name successfully.

See Also

general functions, mktemp(), tmpnam()

Notes

tempnam is not described in the ANSI Standard. Programs that use it will not conform strictly the Standard, and may not be portable to other compilers or environments.

TERM — Environmental Variable

Name the default terminal type
TERM=*terminal type*

The environmental variable **TERM** names the type of terminal that you are using. This variable is read by every program that uses the **termcap** library, to ensure that the correct entry in the file **/etc/termcap** is read when the program is invoked. You should set this variable in your **profile**, to ensure that the system understands what type of terminal you use. The file **/etc/profile** sets **TERM** to **ansipc**.

See Also

environmental variables, me, termcap

termcap — System Maintenance

Terminal-description language
/etc/termcap

termcap is a language for describing terminals and their capabilities. Terminal descriptions are collected in the file **/etc/termcap** and are read by **tgetent** and its related programs to ensure that output to a particular terminal is in a format that that terminal can understand.

A terminal description written in **termcap** consists of a series of fields, which are separated from each other by colons ':'. Every line in the description, with the exception of the last line, must end in a backslash '\'. Tab characters are ignored. Lines that begin with a '#' are comments. A **termcap** description must not exceed 1,024 characters.

The first field names the terminal. Several different names may be used, each separated by a vertical bar '|'; each name given, however, must be unique within the file **/etc/termcap**. By convention, the first listed must be two characters long. The second name is the name by which the terminal is most commonly known; this name may contain no blanks in it.

Other versions of the name may follow. By convention, the last version is the full name of the terminal; here, spaces may be used for legibility. Any of these may be used to name your terminal to the COHERENT system. For example, the name field for the VT-100 terminal is as follows:

```
dl|vt100|vt-100|pt100|pt-100|dec vt100:\
```

Note that the names are separated by vertical bars '|', that the field ends with a colon, and that the line ends with a backslash. Using any of these names in an **export** command will make the correct terminal description available to programs that need to use it.

The remaining fields in the entry describe the *capabilities* of the terminal. Each capability field consists of a two-letter code, and may include additional information. There are three types of capability:

Boolean

This indicates whether or not a terminal has a specific feature. If the field is present, the terminal is assumed to have the feature; if it is absent, the terminal is assumed not to have that feature. For example, the field

am:

is present, **termcap** assumes that the terminal has automatic margins, whereas if that field is not present, the program using **termcap** assumes that the terminal does not have them.

Numeric

This gives the size of some aspect of the terminal. Numeric capability fields have the capability code, followed by a '#' and a number. For example, the entry

co#80:

means that the terminal screen is 80 columns wide.

String capabilities

These give a sequence of characters that trigger a terminal operation. These fields consist of the capability code, an equal sign '=', and the string.

Strings often include escape sequences. A "**\E**" indicates an **<ESC>** character; a control character is indicated with a carat '^' plus the appropriate letter; and the sequences **\b**, **\f**, **\n**, **\r**, and **\t** are, respectively, backspace, formfeed, newline, **<return>**, and tab.

An integer or an integer followed by an asterisk in the string (e.g., '*int**') indicates that execution of the function should be delayed by *int* milliseconds; this delay is termed *padding*. Thus, deletion on lines on the Microterm Mime-2A is coded as:

```
dl=20*^W:
```

dl is the capability code for *delete*, the equal sign introduces the deletion sequence, **20*** indicates that each line deletion should be delayed by 20 milliseconds, and **^W** indicates that the line-deletion code on the Mime-2A is **<ctrl-W>**.

The asterisk indicates that the padding required is proportional to the number of lines affected by the operation. In the above example, the deletion of four lines on the Mime-2A generates a total of 80 milliseconds of padding; if no asterisk were present, however, the padding would be only 20 milliseconds, no matter how many lines were deleted. Also, when an asterisk is used, the number may be given to one decimal place, to show tenths of a millisecond of padding.

Note that with string capabilities, characters may be given as a backslash followed

by the three octal digits of the character's ASCII code. Thus, a colon in a capability field may be given by **\072**. To put a null character into the string, use **\200**, because **termcap** strips the high bit from each character.

Finally, the literal characters '^' and '\' are given by "\^" and "\\".

Capability Codes

The following table lists **termcap**'s capability codes. **Type** indicates whether the code is boolean, numeric, or string; a dagger '†' indicates that this capability may include padding, and a dagger plus an asterisk "†*" indicates that it may be used with the asterisk padding function described above.

Name	*Type*	*Definition*
ae	string†	End alternate set of characters
al	string†*	Add blank line
am	boolean	Automatic margins
as	string†	Start alternate set of characters
bc	string	Backspace character, if not **^H**
bs	boolean	Backspace character is **^H**
bt	string†	Backtab
bw	boolean	Backspace wraps from column 0 to last column
CC	string	Command character in prototype if it can be set at terminal
cd	string†*	Clear to end of display
ce	string†	Clear line
ch	string†	Horizontal cursor motion
cl	string†*	Clear screen
cm	string†	Cursor motion, both vertical and horizontal
co	number†	Number of columns
cr	string†*	**<return>**; default **^M**
cs	string†	Change scrolling region (DEC VT100 only); resembles **cm**
cv	string†	Vertical cursor motion
da	boolean†	Display above may be retained
dB	number	Milliseconds of delay needed by **bs**
db	boolean	Display below may be retained
dC	number	Milliseconds of delay needed by **cr**
dc	string†*	Delete a character
dF	number	Milliseconds of delay needed by**ff**
dl	string†*	Delete a line
dm	string	Enter delete mode
dN	number	Milliseconds of delay needed by **nl**
do	string	Move down one line
dT	number	Milliseconds of delay needed by tab
ed	string	Leave delete mode
ei	string	Leave insert mode; use **:ei=:** if this string is the same as **ic**
eo	string	Erase overstrikes with a blank
ff	string†*	Eject hardcopy terminal page; default **^L**
hc	boolean	Hardcopy terminal
hd	string	Move half-line down, i.e., forward 1/2 line feed)

ho	string	Move cursor to home position; use if **cm** is not set
hu	string	Move half-line up, i.e., reverse 1/2 line feed
hz	string	Cannot print tilde '~' (Hazeltine terminals only)
ic	string†	Insert a character
if	string	Name of the file that contains **is**
im	string	Begin insert mode; use **:im=:** if **ic** has not been set
in	boolean	Nulls are distinguished in display
ip	string†*	Insert padding after each character listed
is	string	Initialize terminal
k0-k9	string	Codes sent by function keys 0-9
kb	string	Code sent by backspace key
kd	string	Code sent by down-arrow key
ke	string	Leave "keypad transmit" mode
kh	string	Code sent by home key
kl	string	Code sent by left-arrow key
kn	number	No. of function keys; default is 10
ko	string	Entries for for all other non-function keys
kr	string	Code sent by right-arrow key
ks	string	Begin "keypad transmit" mode
ku	string	Code sent by up-arrow key
l0-l9	string	Function keys labels if not f0-f9
li	number	Number of lines
ll	string	Move cursor to first column of last line; use if **cm** is not set
ma	string	Map keypad-to-cursor movement for **vi** version 2
mi	boolean	Cursor may be safely moved while in insert mode
ml	string	Turn on memory lock for area of screen above cursor
ms	boolean	Cursor may be safely moved while in standout or underline mode
mu	string	Turn off memory lock
nc	boolean	**<return>** does not work
nd	string	Move cursor right non-destructively
nl	string†*	Newline character; default is **\n**. *Obsolete*
ns	boolean	Terminal is CRT, but does not scroll
os	boolean	Terminal can overstrike
pc	string	Pad character any character other than null
pt	boolean	Terminal's tabs set by hardware; may need to be set with **is**
se	string	Exit standout mode
sf	string†	Scroll forward
sg	number	Blank characters left by **so** or **se**
so	string	Enter standout mode
sr	string†	Reverse scroll
ta	string†	Tab character other than **^I**, or used with character padding

tc	string	Similar terminal — must be last field in entry
te	string	End a program that uses **cm**
ti	string	Begin a program that uses **cm**
uc	string	Underscore character and skip it
ue	string	Leave underscore mode
ug	number	Blank characters left by **us** or **ue**
ul	boolean	Terminal underlines but does not overstrike
up	string	Move up one line
us	string	Begin underscore mode
vb	string	Visible bell; may not move cursor
ve	string	Exit open/visual mode
vs	string	Begin open/visual mode
xb	boolean	Beehive terminal (f1=**<esc>**, f2=**<crtl-C>**)
xn	boolean	Newline is ignored after wrap
xr	boolean	**<return>** behaves like **ce \r \n**
xs	boolean	Standout mode is not erased by writing over it
xt	boolean	Tabs are destructive

The following is the **termcap** description of the Zenith Z-19 terminal. The meaning of each field will be described:

```
kb|h19|heath|h19b|heathkit|heath-19|z19|zenith|heathkit h19:\
        :al=1*\EL:am:bs:cd=\EJ:ce=\EK:cl=\EE:cm=\EY%+ %+ :\
        :co#80:dc=\EN:dl=1*\EM:do=\EB:ei=\EO:ho=\EH:\
        :im=\E@:li#24:mi:nd=\EC:as=\EF:ae=\EG:ms:pt:\
        :sr=\EI:se=\Eq:so=\Ep:up=\EA:vs=\Ex4:ve=\Ey4:\
        :kb=^h:ku=\EA:kd=\EB:kl=\ED:kr=\EC:kh=\EH:kn#8:\
        :k1=\ES:k2=\ET:k3=\EU:k4=\EV:k5=\EW:\
        :l6=blue:l7=red:l8=white:k6=\EP:k7=\EQ:k8=\ER:
```

The first field, which occupies line 1, gives the various aliases for this terminal. The Heathkit H-19, which appears most prominently, was the home-kit version of the commercially sold Z-19. The remaining fields mean the following:

:al=1*\EL:	**<esc>L** adds new blank line; use one millisecond for each line added
:am:	Terminal has automatic margins
:bs:	Backspace character is **<ctrl>-H** (the default)
:cd=\EJ:	**<esc>J** clears to end of display
:ce=\EK:	**<esc>K** clears to end of line
:cl=\EE:	**<esc>E** clears screen
:cm=\EY%+ %+ :	Cursor motion (described later)
:co#80:	Screen has 80 columns
:dc=\EN:	**<esc>N** deletes a character (backslash indicates end of a line)
:dl=1*\EM:	**<esc>M** deletes a line
:do=\EB:	**<esc>B** moves cursor down one line
:ei=\EO:	**<esc>O** exits from insert mode
:ho=\EH:	**<esc>H** moves cursor to home position
:im=\E@:	**<esc>@** begins insert mode (note that **ic** is set)
:li#24:	Terminal has 24 lines
:mi:	Cursor may be moved safely while terminal

	is in insert mode
:nd=\EC:	**<esc>C** moves cursor right non-destructively
:as=\EF:	**<esc>F** begins set of alternate characters
:ae=\EG:	**<esc>G** ends set of alternate characters
:ms:	Cursor may be moved safely while terminal is in standout and underline mode
:pt:	Terminal has hardware tabs
:sr=\EI:	**<esc>I** reverse-scrolls the screen
:se=\Eq:	**<esc>q** exits standout mode
:so=\Ep:	**<esc>p** begins standout mode
:up=\EA:	**<esc>A** moves the cursor up one line
:vs=\Ex4:	**<esc>x** begins visual mode; insert 4 milliseconds of padding when visual mode is begun
:ve=\Ey4:	**<esc>y** ends visual mode; insert 4 milliseconds of padding when visual mode is ended
:kb=^h:	Backspace key sends **<Ctrl>-H**
:ku=\EA:	Up-arrow key sends **<esc>A**
:kd=\EB:	Down-arrow key sends **<esc>B**
:kl=\ED:	Left-arrow key sends **<esc>D**
:kr=\EC:	Right-arrow key sends **<esc>C**
:kh=\EH:	Home key sends **<esc>H**
:kn#8:	There are eight other keys on the keyboard
:k1=\ES:	Other key 1 sends **<esc>S**
:k2=\ET:	Other key 2 sends **<esc>T**
:k3=\EU:	Other key 3 sends **<esc>U**
:k4=\EV:	Other key 4 sends **<esc>V**
:k5=\EW:	Other key 5 sends **<esc>W**
:l6=blue:	Other key 6 is labeled "blue"
:l7=red:	Other key 7 is labeled "red"
:l8=white:	Other key 8 is labeled "white"
:k6=\EP:	Other key 6 sends **<esc>P**
:k7=\EQ:	Other key 7 sends **<esc>Q**
:k8=\ER:	Other key 8 sends **<esc>R**

Note that the last field did *not* end with a backslash; this indicated to the COHERENT system that the **termcap** description was finished.

A terminal description does not have to be nearly so detailed. If you wish to use a new terminal, first check the following table to see if it already appears by **termcap**. If it does not, check the terminal's documentation to see if it mimics a terminal that is already in **/etc/termcap**, and use that description, modifying it if necessary and changing the name to suit your terminal. If you must create an entirely new description, first prepare a skeleton file that contains the following basic elements: number of lines, number of columns, backspace, cursor motion, line delete, clear screen, move cursor to home position, newline, move cursor up a line, and non-destructive right space. For example, the following is the **termcap** description for the Lear-Siegler ADM-3A terminal:

```
la|adm3a|3a|lsi adm3a:\
        :am:bs:cd=^W:ce=^X:cm=\E=%+ %+ :cl=^Z:co#80:ho=^^:li#24:\
        :nd=^L:up=^K:
```

Once you have installed and debugged the skeleton description, add details gradually until every feature of the terminal is described.

Cursor Motion

The cursor motion characteristic contains **printf**-like escape sequences not used elsewhere. These encode the line and column positions of the cursor, whereas other characters are passed unchanged. If the **cm** string is considered as a function, then its arguments are the line and the column to which the cursor is to move; the **%** codes have the following meanings:

%d Decimal number, as in **printf**. The origin is 0.

%2 Two-digit decimal number. The same as **%2d** in **printf**.

%3 Three-digit decimal number. The same as **%3d** in **printf**.

%. Single byte. The same as **%c** in **printf**.

%+*n* Add *n* to the current position value. *n* may be either a number or a character.

%>*nm* If the current position value is greater than *n*+*m*; then there is no output.

%r Reverse order of line and column, giving column first and then line. No output.

%i Increment line and column.

%% Give a **%** sign in the string.

%n Exclusive or line and column with 0140 (Datamedia 2500 terminal only).

%B Binary coded decimal (16 * (*n*/10))+(*n*%10). No output.

%D Reverse coding (*n*-(**2***(*n***%16**)). No output (Delta Data terminal only).

To send the cursor to line 3, column 12 on the Hewlett-Packard 2645, the terminal must be sent **<esc>&a12c03Y** padded for 6 milliseconds. Note that the column is given first and then the line, and that the line and column are given as two digits each. Thus, the **cm** capability for the Hewlett-Packard 2645 is given by:

```
:cm=6\E&%r%2c%2Y:
```

The Microterm ACT-IV needs the current position sent preceded by a **<Ctrl-T>**, with the line and column encoded in binary:

```
:cm=^T%.%.:
```

Terminals that use **%.** must be able to backspace the cursor (**bs** or **bc**) and to move the cursor up one line on the screen (**up**). This is because transmitting **\t**, **\n**, **\r**, or **<ctrl-D>** may have undesirable consequences or be ignored by the system.

Similar Terminals

If your system uses two similar terminals, one can be defined as resembling the other, with certain exceptions. The code **tc** names the similar terminal. This field must be *last* in the **termcap** entry, and the combined length of the two entries cannot exceed 1,024 characters. Capabilities given first over-ride those in the similar terminal, and capabilities in the similar terminal can be cancelled by **xx@** where *xx* is the capability. For example, the entry

```
hn|2621nl|HP 2621nl:ks@:ke@:tc=2621
```

defines a Hewlett-Packard 2621 terminal that does not have the **ks** and **ke** capabilities, and thus cannot turn on the function keys when in visual mode.

Initialization

A terminal initialization string may be given with the **is** capability; if the string is too long, it may be read from a file given by the **if** code. Usually, these strings set the tabs on a terminal with settable tabs. If both **is** and **if** are given, **is** will be printed first to clear the tabs, then the tabs will be set from the file specified by **if**. The Hewlett-Packard 2626 has:

```
:is=\E&j@\r\E3\r:if=/usr/lib/tabset/stdcrt:
```

Terminals Supported

The following table lists the terminals described in **/etc/termcap**, and an abbreviated name for each.

Name	*Terminal*
act5	Microterm Act V
adm3a	Lear-Siegler ADM3A
adm31	Lear-Siegler ADM31
ansipc	AT COHERENT console
cohibm	PC COHERENT console
dos	DOS 3.1 ANSI.SYS
h1510	Hazeltine 1510
h19	Heathkit H-19
h19a	Heathkit H-19 in ANSI
mime1	Microterm Mime1
mime2a	Microterm Mime2a
mime3a	Microterm Mime3a
qvt102	Qume QVT-102
qume5	Qume Sprint 5
tvi912	Televideo 920
tvi920	Televideo 920
tvi925	Televideo 925
vt52	DEC VT-52
vt100	DEC VT-100
vt100n	DEC VT-100 without initialization
vt100s	DEC VT-100, 132 columns, 14 lines
vt100w	DEC VT-100, 132 columns, 24 lines
wy50	Wyse 50

Files

/etc/termcap

See Also

modemcap, system maintenance, terminal, terminal-independent operations

terminal — Technical Information

This article describes how you can hook up a terminal to your COHERENT system via a serial port. It also discusses common problems that arise with this procedure, as diagnosed daily by the technical support staff at Mark Williams Company. For information on connecting a modem to your computer's serial port, see the article **modem**.

Hooking Up a Terminal to COHERENT

This process is straightforward, but can be confusing if you overlook any details. Typical problems include send/receive confusion, baud rate confusion, and shell/no shell confusion.

Send/Receive Confusion

A serial connection between your computer and a terminal requires at least three wires: one each for pins 2, 3, and 7. These pins, respectively, control send (TD), receive (RD), and signal-ground (Gnd or SG). These pin numbers correspond to the 25-pin "DB-25" connectors used on most equipment. If your system has the AT-style nine-pin "DB-9" connectors, you will need to wire to the corresponding signals. See the Lexicon entry for **RS-232** for details of the pin-outs for these two connectors.

When hooking up a terminal to a serial port using a three-wire connection, you must cross pins 2 and 3, so that each device's send pin talks to the other device's receive pin. You can plug a device called a "null modem" between the cable and the serial port, to do this automatically. Unless someone has sat down and taught you how to solder connectors, we strongly urge you to purchase the necessary cable and null modem at your local computer store or electronics shop.

Note that the only symptom of a problem in the cable is that nothing appears on your terminal when you type.

Baud-Rate Confusion

The terminal and the computer must speak to each other at the same *baud rate*. A typical symptom of baud-rate confusion is garbage characters on the screen. When the wiring is wrong, you see nothing; when the baud rate is wrong, you see something, but nothing meaningful. You can fix baud-rate problems by using the command **stty** to reset the baud rate on the port, or resetting the baud rate on the terminal. For directions on how to reset the baud rate for a port, see the Lexicon entry for **stty**.

The Old Shell Game

Before a terminal is useful to you, you must *enable* the port into which it is plugged. Enabling a port means that the COHERENT system creates a shell for that port: this, in turn, means that COHERENT prints a login prompt on the device plugged into that port, and reads and processes interactively commands that are entered from that port. The COHERENT system also restricts permissions on all enabled serial ports, so that only the superuser **root** can read and write to the port. This prevents other users who may be using the system from accessing the serial port.

Note that not all ports need be enabled: printer ports, for example, should not be enabled; nor should you enable any port whose device you want to accept data passively.

When you boot the COHERENT system, it reads system file **/etc/ttys** and creates a shell for each serial port that needs one. One way to enable a port is to log in as the superuser **root**, then use a text editor to change the port's entry in **/etc/ttys**, as described its Lexicon article. Finally, typing the command

```
kill quit 1
```

forces COHERENT to re-read **/etc/ttys** and so create a shell for the port. Note that doing this will ensure that the port is re-enabled every time you boot.

A better way to enable a port is to use the command **enable**, as described in its Lexicon article. For example, to put up a shell on COM port **/dev/com1r**, log in as the superuser **root** and type the command:

```
/etc/enable com1r
```

Exiting Raw Mode

A terminal is in *cooked* mode. In cooked mode, the tty driver interprets and correctly processes such predefined characters as the end-of-file character or the quit character. In *raw* mode, however, processing of such characters is turned off; and in general the terminal will behave bizarrely. Raw mode is used by programs that do not want the tty driver to interpret characters; for example, a program that uses a tty to transmit a binary to another machine does not want the tty driver to be interpreting the binary information being passed through it.

Occasionally, a program will exit abruptly and leave the terminal in raw mode. To return to cooked mode, use the command **<ctrl-J> stty sane <ctrl-J>**. This invokes the command **stty**, which lets you manipulate terminal settings, to restore the previous cooked state. See the Lexicon entry on **stty** for details on raw and cooked modes; this article also describes the options of this most useful command.

See Also

modem, RS-232, sgtty, stty, technical information, termcap, ttys

Notes

One final bit of hard-won wisdom: once you have something working, write down what you did, and store it in a place where you won't lose it. Note especially what connectors are where and how they have been cabled together. It makes life easier just knowing that you are looking for a female-to-female cable instead of male-to-female or male-to-male. If you know whether to insert a null modem, you are even better off. @.

terminal-independent operations — Overview

The COHERENT system includes a set of functions, found in the library **/usr/lib/libterm.a**, that extract and use the descriptions stored in the file **/etc/termcap**. These functions return information about how a given terminal functions; thus, they allow a program to address any number of different terminals correctly, without having to change source code or recompile.

The following functions perform terminal-independent operations:

tgetent()	Read the appropriate **termcap** entry.
tgetflag()	Check if a given Boolean capability is present in the terminal's entry.
tgetnum()	Return the value of a numeric **termcap** feature.
tgetstr()	Read and decode a **termcap string feature.**
tgoto()	Read and decode a **termcap** cursor-addressing string.
tputs()	Read and decode the leading padding information of a **termcap** string feature.

See the Lexicon entry for each function for more details on its operation.

The external variable **ospeed** is the output speed to the terminal as encoded by **stty**. The external variable *PC* is a padding character from the **pc** capability if a null (**<crtl-@>**) is not appropriate.

Files

/etc/termcap — Terminal capabilities data base
/usr/lib/libterm.a — Function library

See Also

libraries, stty, termcap, tgetent(), tgetflag(), tgetnum(), tgetstr(), tgoto(), tputs()

termio.h — Header File

Definitions used with terminal input and output
#include <termio.h>

termio.h defines structures and constants used by functions that control terminal input and output.

See Also

header files, termio

test — Command

Evaluate conditional expression
test *expression* ...

test evaluates an *expression*, which consists of string comparisons, numerical comparisons, and tests of file attributes. For example, a **test** command might be used within a shell command file to test whether a certain file exists and is readable. The logical result (true or false) of the *expression* is returned by the command, for use by a shell construct such as **if**.

expression is constructed from the following elements, which are true if the given condition holds and false if not:

exp1 **-a** *exp2*	Both expressions *exp1* and *exp2* are true.
-b *file*	*file* is a block-special device. **ksh** only.
-c *file*	*file* is a character-special file. **ksh** only.
-d *file*	*file* exists and is a directory.
file1 **-ef** *file2*	*file1* is the same file as *file2*. **ksh** only.
n1 **-eq** *n2*	Numbers *n1* and *n2* are equal.
-f *file*	*file* exists and is not a directory.
-g *file*	File mode has **setgid** bit. **ksh** only.
n1 **-ge** *n2*	Number *n1* is greater than or equal to *n2*.
n1 **-gt** *n2*	Number *n1* is greater than *n2*.
-k *file*	File mode has sticky bit. **ksh** only.
-L *file*	File is a symbolic link. **ksh** only.
n1 **-le** *n2*	Number *n1* is less than or equal to *n2*.
n1 **-lt** *n2*	Number *n1* is less than *n2*.
-n *string*	*string* has nonzero length.
n1 **-ne** *n2*	Numbers *n1* and *n2* are not equal.
file1 **-nt** *file2*	*file1* is newer than *file2*. **ksh** only.
exp1 **-o** *exp2*	Either expression *exp1* or *exp2* is true. **-a** has greater precedence than **-o**.

file1 **-ot** *file2* — *file1* is older than *file2*. **ksh** only.

-p *file* — *file* is a named pipe. **ksh** only.

-r *file* — *file* exists and is readable.

-s *file* — *file* exists and has nonzero size.

-t *[fd]* — *fd* is the file descriptor number of a file which is open and a terminal. If no *fd* is given, it defaults to the standard output (file descriptor 1).

-u *file* — File mode has **setuid** set. **ksh** only.

-w *file* — *file* exists and is writable.

-x *file* — *file* exists and executable. **ksh** only.

-z *string* — *string* has zero length (is a null string).

string — *string* has nonzero length.

string1 **=** *string2* — *string1* is equal to *string2*.

! *exp* — Negates the logical value of expression *exp*.

string1 **!=** *string2* — *string1* is not equal to *string2*.

(*exp*) — Parentheses allow expression grouping.

Example

The following example uses the **test** command to determine whether a file is writable.

```
if test ! -w /dev/lp
then
     echo The line printer is inaccessible.
fi
```

Under COHERENT, the command '[' is linked to **test**. If invoked as '[', **test** checks that its last argument is ']'. This allows an alternative syntax: simply enclose *expression* in square brackets. For example, the above example can be written as follows:

```
if [ ! -w /dev/lp ]
then
     echo The line printer is inaccessible.
fi
```

For a more extended example of the square-bracket syntax, see **sh**.

See Also

commands, expr, find, if, ksh, sh, while

Notes

The Korn shell's version of this command is based on the public-domain version written by Erik Baalbergen and Arnold Robbins.

tgetent() — Terminal-Independent Operation

Read termcap entry
int tgetent(*bp, name*)
char **bp*, **name*;

tgetent is one of a set of functions that help COHERENT to perform terminal-independent operations. It extracts the entry from file **/etc/termcap** for the terminal *name* and writes it into a buffer at address *bp*. *bp* should be a character buffer of 1,024 bytes and must be retained through all subsequent calls to the other functions. It returns **-1** if it cannot open **/etc/termcap**, zero if the terminal name given does not have an entry, and one upon a successful search.

tgetent first looks in the environment to see if the **termcap** variable had already been set. If it finds that the variable **termcap** has been set, that the value does *not* begin with a slash, and that the terminal type name in the **termcap** variable is the same as that in the environment variable **TERM**, then **tgetent** uses the **termcap** string instead of reading the file **/etc/termcap**. However, if the **termcap** string does begin with a slash, then it is used as the path name of a terminal-capabilities file other than **/etc/termcap.** This can speed entry into programs that call **tgetent**, and can be used to help debug new terminal descriptions.

Files

/etc/termcap — Terminal capabilities data base
/usr/lib/libterm.a — Function library

See Also

termcap, terminal-independent operation

tgetflag() — Terminal Independent Operation

Get termcap Boolean entry
int tgetflag(*name*)
char **name*;

tgetflag is one of a set of functions that help COHERENT to perform terminal-independent operation. It returns one if the requested Boolean capability *name* is present in the terminal's **termcap** entry, zero if it is not.

Files

/etc/termcap — Terminal capabilities data base
/usr/lib/libterm.a — Function library

See Also

termcap, terminal-independent operation

tgetnum() — Terminal-Independent Operation

Get termcap numeric feature
int tgetnum(*name*)
char **name*;

tgetnum is one of a set of functions that permit the COHERENT system to perform terminal-independent operations. It returns the value of the numeric feature *name*, as defined in the terminal's **termcap** entry. It returns **-1** if the feature is not present in the terminal's entry.

Files

/etc/termcap — Terminal capabilities data base
/usr/lib/libterm.a — Function library

See Also

termcap, terminal-independent operations

tgetstr() — Terminal-Independent Operation

Get termcap string entry
char *tgetstr(*name, area***)**
char *****name*, *******area*;

tgetstr is one of a set of functions that help COHERENT to perform terminal-independent operations. It reads the string value of feature *name* from the terminal's **termcap** description, and writes it into the buffer at address *area*. It also advances the value of the pointer to *area*.

tgetstr decodes the abbreviations for the fields used in the **termcap** entry, except for padding and for cursor-addressing information.

Files

/etc/termcap — Terminal capabilities data base
/usr/lib/libterm.a — Function library

See Also

termcap, terminal-independent operation

tgoto() — Terminal-Independent Operation

Read/interpret termcap cursor-addressing string
char *tgoto(*cm, destcol, destline***)**
char ****cm*; **int *scrcol*, *scrline*;

tgoto is one of a set of funtions that permit COHERENT to perform terminal-independent operations. It decodes a cursor-addressing string from the *cm* **termcap** feature, and writes it onto the screen, at column *scrcol* and line *destline*. **tgoto** uses the external variables *UP* (from the **up** feature) and *BC* (if **bc** is given rather than **bs**) if it is necessary to avoid placing **\n**, **<ctrl-D>**, or **<ctrl-@>** into the returned string. Programs calling **tgoto** should turn off the **XTABS** bits, as **tgoto** may write a tab. If a '%' sequence is given that is not understood, **tgoto** returns "OOPS".

Files

/etc/termcap — Terminal capabilities data base
/usr/lib/libterm.a — Function library

See Also

termcap, terminal-independent operation

time() — Time Function (libc)

Get current time
#include <time.h>
#include <sys/types.h>
time_t time(*tp***) time_t ****tp*;

time reads the current system time. *tp* points to a data element of the type **time_t**, which is defined in the header file **types.h** as being equivalent to a **long**. COHERENT defines the current system time as the number of seconds since January 1, 1970, 0h00m00s GMT.

Example

For an example of this function, see the entry for **asctime**.

See Also

date, ftime, time (overview)

time — Overview

COHERENT includes a number of routines that allow you to set and manipulate time, as recorded on the system's clock, into a variety of formats. These routines should be adequate for nearly any task that involves temporal calculations or the maintenance of data gathered over a long period of time.

All functions, global variables, and manifest constants used in connection with time are defined and described in the header files **time.h** and **timeb.h**.

The COHERENT system includes the following functions to manipulate time:

asctime	Convert time structure to ASCII string
ctime	Convert system time to an ASCII string
ftime	Get the current time from the operating system
gmtime	Convert system time to calendar structure
localtime	Convert system time to calendar structure
settz	Set local time zone
time	Get current time

To print out the local time, a program must perform the following tasks: First, read the system time with **time**. Then, it must pass **time**'s output to **localtime**, which breaks it down into the **tm** structure. Next, it must pass **localtime**'s output to **asctime**, which transforms the **tm** structure into an ASCII string. Finally, it must pass the output of **asctime** to **printf**, which displays it on the standard output device. See the entry for **asctime** for an example of such a program.

Example

For an example of time functions, see the entry for **asctime**.

See Also

Libraries

time — Command

Time the execution of a command
time [*command*]

time invokes the given *command* with any arguments provided. Upon termination, **time** prints the elapsed real time, CPU time in the system, and CPU time in the user program on the standard error output.

If no *command* is given, **time** simply invokes **date** to print the current time of day.

See Also

commands, date, ps, times

Diagnostics

If the *command* terminates abnormally, the reason is printed.

time.h — Header File

Give time-description structure
#include <time.h>

time.h is a header file that contains descriptions and declarations for elements used to manipulate system time under COHERENT.

See Also

header files, time

timeb.h — Header File

Declare timeb structure
#include <sys/timeb.h>

The header file **timeb.h** declares the structure **timeb**, which is used by the function **ftime** to return time information.

See Also

ftime(), header files, time

timef.h — Header File

Definitions for user-level timed functions
#include <timef.h>

timef.h defines structures and constants used by user-level timed functions.

See Also

header files

timeout.h — Header File

Define the timer queue
#include <timeout.h>

timeout.h defines the timeout queue. The timeout queue can, as its name implies, be used to call a function when a process has "timed out".

See Also

header files

times() — COHERENT System Call

Obtain process execution times
#include <sys/times.h>
#include <sys/const.h>
int times(*tbp***)**
struct tbuffer ****tbp*;**

times reads CPU time information about the current process and its children, and writes it into the structure pointed to by *tbp*. The structure **tbuffer** is declared in the header file **sys/times.h**, as follows:

```
struct tbuffer {
        long   tb_utime;     /* process user time */
        long   tb_stime;     /* process system time */
        long   tb_cutime;    /* childrens' user times */
        long   tb_cstime;    /* childrens' system times */
};
```

All of the times are measured in basic machine cycles, or **HZ**, which may be obtained from the header file **sys/const.h**. Under AT COHERENT, **HZ** is 100.

The childrens' times include the sum of the times of all terminated child processes of the current process and of all of their children. The *user* time represents execution time of user code, whereas *system* time represents system overhead, such as executing system calls, processing signals, and other monitoring functions.

Files

<sys/times.h>
<sys/const.h>

See Also

acct(), COHERENT system calls, const.h, ftime(), time()

times — Command

Print total user and system times
times

times prints the total elapsed user time and system time for the current shell and all its children. It gives each time in minutes, seconds and tenths of seconds. For example,

```
1m11.8s 1m35.8s
```

indicates a total user time of 1 minute 11.8 seconds, and a total system time of 1 minute 35.8 seconds.

The shell executes **times** directly.

See Also

commands, ksh, time, sh

times.h — Header File

Definitions used with times() system call
#include <times.h>

times.h defines the structure **tbuffer**, which is used to implement the **times** system call.

See Also

header files, times()

TIMEZONE — Environmental Variable

Time zone information
TIMEZONE=*standard*:*offset*[:*daylight*: *date*:*date*:*hour*:*minutes*]

TIMEZONE is an environmental parameter that holds information about the user's time zone. This information is used by COHERENT's time routines to construct their description of the current time and day.

To set the **TIMEZONE** parameter, use the following command:

```
export TIMEZONE=[description]
```

where [description] is the string that describes your time zone. What this string consists of will be described below. Most users write this command into the file **.profile**, so that **TIMEZONE** is set automatically whenever they log onto the COHERENT system.

COHERENT's installation procedure creates file **/etc/timezone**, which sets **TIMEZONE**. This file is executed by **/etc/profile** when each user logs in. Thus, you must set the **TIMEZONE** in your **.profile** only if it differs from the system's **TIMEZONE** as set in **/etc/timezone**. This would be necessary if, for example, a user in New York were to regularly login on a system in Chicago.

The Description String

A **TIMEZONE** description string consists of seven fields that are separated by colons. Fields 1 and 2 must be filled; fields 3 through 7 are optional.

Field 1 gives the name of your standard time zone. Field 2 gives the time zone's offset from Greenwich Mean Time in minutes. Offsets are positive for time zones west of Greenwich and negative for time zones east of Greenwich. For example, users in Chicago set these fields as follows:

```
TIMEZONE=CST:360
```

CST is an abbreviation for Central Standard Time, that area's time zone; and 360 refers to the fact that Chicago's time zone is 360 minutes (six hours) behind that of Greenwich.

Field 3 gives the name of the local daylight saving time zone. In Chicago, for example, this field would be set as follows:

```
TIMEZONE=CST:360:CDT
```

CDT is an abbreviation for Central Daylight Time. The absence of this field indicates that your area does not use daylight saving time.

Fields 4 and 5 specify the dates on which daylight saving time begins and ends. If field 3 is set but fields 4 and 5 are not, changes between standard time and daylight saving time are assumed to occur at the times legislated in the United States: at 2 A.M. standard time on the first Sunday in April, and at 2 A.M. daylight saving time on the last Sunday in October.

Fields 4 and 5 each consist of three numbers separated by periods. The first number specifies which occurrence of the day in the month marks the change, counting positive occurrences from the beginning of the month and negative occurrences from the the end of the month. The second number specifies a day of the week, numbering Sunday as one. The third number specifies a month of the year, numbering January as one. For example, in Chicago fields 4 and 5 are set to the following:

```
TIMEZONE=CST:360:CDT:1.1.4:-1.1.10
```

If the first number in either field is set to zero, then the last two numbers are assumed to indicate an absolute date. This is done because some countries switch to daylight saving time on the same day each year, instead of a given day of the week.

Finally, fields 6 and 7 specify the hour of the day at which daylight saving time begins and ends, and the number of minutes of adjustment. In Chicago, these are set as follows:

```
TIMEZONE=CST:360:CDT:1.1.4:-1.1.10:2:60
```

The '2' of field 6 indicates that the switch to daylight savings time occurs at 2 A.M. The "60" of field 7 indicates that daylight savings time changes the local time by 60 minutes. Although 60 minutes is the standard change, some regions of the world shift by 30, 45, 90,

or 120 minutes; the last shift is also called "double daylight saving time".

For an example of this variable's use in a program, see the entry for **asctime**.

See Also

environmental variables, time (overview)

Notes

For those requiring more information on this subject, much research has been performed by astrologers. See *Time Changes in the World*, compiled by Doris Chase Doane (three volumes, Hollywood, California, Professional Astrologers, Inc., 1970).

tmpnam() — General Function (libc)

Generate a unique name for a temporary file
#include <stdio.h>
char *tmpnam(*name***);**
char ***name;**

tmpnam constructs a unique name for a file. The names returned by **tmpnam** generally are mechanical concatenations of letters, and therefore are mostly used to name temporary files, which are never seen by the user. A file named by **tmpnam** does not automatically disappear when the program exits. You must explicitly remove it before the program ends if you want it to disappear.

name points to the buffer into which **tmpnam** writes the name it generates. If *name* is set to NULL, **tmpnam** writes the name into an internal buffer that may be overwritten each time you call this function.

tmpnam returns a pointer to the temporary name. Unlike the related function **tempnam**, **tmpnam** assumes that the temporary file will be written into directory **/tmp** and builds the name accordingly.

Example

For an example of this function, see **execve**.

See Also

general functions, mktemp(), STDIO, tempnam()

Notes

If you want the file name to be written into *buffer*, you should allocate at least **L_tmpnam** bytes of memory for it; **L_tmpnam** is defined in the header **stdio.h**.

tolower() — ctype Macro

Convert characters to lower case
#include <ctype.h>
int tolower(*c***) int** *c***;**

tolower converts the letter *c* to lower case. **tolower** returns **c** converted to lower case.

Note that **tolower** is not guaranteed to work correctly if handed anything other than an upper-case character, that is, a character for which **isupper()** returns true.

Example

The following example demonstrates **tolower** and **toupper**. It reverses the case of every character in a text file.

```
#include <ctype.h>
#include <stdio.h>

main()
{
	FILE *fp;
	int ch;
	int filename[100];

	printf("Enter name of file to use: ");
	fflush(stdout);
	gets(filename);

	if ((fp = fopen(filename,"r")) != NULL) {
		while ((ch = fgetc(fp)) != EOF) {
			if (islower(ch))
				putchar(toupper(ch));
			else if (isupper(ch))
				putchar(tolower(ch));
			else
				putchar(ch);
		}
	} else
		printf("Cannot open %s.\n", filename);
}
```

See Also

ctype, toupper()

touch — Command

Update modification time of a file
touch [-c] *file* ...

COHERENT keeps track of when each file was last modified. **touch** changes the modification time of each *file* to the current time, but does not modify its contents. By default, **touch** creates *file* if it does not already exist; the **-c** flag suppresses this.

See Also

commands, make

toupper() — ctype macro

Convert characters to upper case
#include <ctype.h>
int toupper(*c***) int** *c***;**

toupper is a macro that converts the letter *c* to upper case and returns the converted character.

Note that **toupper** is not guaranteed to work correctly if it is passed something other than a lower-case character, that is, any character for which **islower()** returns true.

Example

For an example of this routine, see the entry for **tolower**.

See Also

ctype, tolower()

tputs() — Terminal-Independent Operation

Read/decode leading padding information
tputs(*cp, affcnt, outc*)
register char **cp*; **int** *affcnt*; **int** (**outc*)();

tputs is one of a set of functions that permit COHERENT to perform terminal-independent operations. It decodes the leading padding information of the string *name*. *affcnt* is the number of lines affected by the operation, and is set to one if it is not applicable. *outc* is a routine called to write each character.

Files

/etc/termcap — Terminal capabilities data base
/usr/lib/libterm.a — Function library

See Also

termcap, terminal-independent operation

tr — Command

Translate characters
tr [-cds] *string1* [*string2*]

tr reads characters from the standard input, possibly translates each to another value or deletes it, and writes to standard output.

Each specified *string* may contain literal characters of the form *a* or *b* (where *b* is non-numeric), octal representations of the form *ooo* (where *o* is an octal digit), and character ranges of the form *X-Y*. **tr** rewrites each *string* with the appropriate conversions and range expansions.

If an input character is in *string1*, **tr** outputs the corresponding character of *string2*. If *string2* is shorter than *string1*, the result is the last character in *string2*.

The following flags control how **tr** translates characters:

-c Replace *string1* by the set of characters not in *string1*.

-d Delete characters in *string1* rather than translating them.

-s The "squeeze" option: map a sequence of the same character from *string1* to one output character.

Example

The following example prints all sequences of four or more spaces or printing characters from **infile**:

```
tr -cs ' -~' '\12' <infile | grep ....
```

Here *string1* is the range from **<space>** to '~', which includes all printing characters. Because this example uses the flags **-cs**, **tr** maps sequences of nonprinting characters to newline (octal 12).

See Also

ASCII, commands, ctype, sed

trap — Command

Execute command on receipt of signal
trap [*command*] [*n* ...]

trap instructs the shell to execute the given *command* when the shell receives signal *n* or any other signal in the optional list. If the *command* is omitted, **trap** resets traps for the given signals to the original values. If the *command* is a null string (i.e., a string that consists only of one null character), the shell ignores the given signals. If *n* is zero, the shell executes the specified *command* when it exits. When it is invoked with no arguments, **trap** prints the signal number and command for each signal on which a trap is set.

The shell executes **trap** directly.

Example

The following example takes two files and outputs only those lines which are the same.

```
# If input only one file-name then simply "cat".
if [ $# = 1 ]; then
     cat $1
     exit 0

# If input two file-names - Ok, else "Usage".
else
     if [ $# != 2 ]; then
          echo "Usage: cmn file1 [file2]"
          exit 1
     fi
fi

# TMP is original name of temporary file (/tmp/temp_(pid)
TMP=/tmp/temp_$$

# Temporary file has to be removed
trap 'rm $TMP; exit 1' 1 2 9

# Difference between "file1" and "difference between file1 and file2"
# is the common strings "file1" and "file2"
# The strings that are in "file1" and absent in "file2" print in TMP.
diff $1 $2 | sed -n -e "s/^< //p" > $TMP

# The strings that are in "file1" and absent in TMP print in stdout.
diff $1 $TMP | sed -n -e "s/^< //p"

# Remove temporary file
rm $TMP
```

See Also

commands, ksh, sh, signal

troff — Command

Extended text-formatting language
troff [*option* ...] [*file* ...]

The command **troff** is the COHERENT typesetter and text-formatting language. It performs typeset-quality text formatting, suitable for printing on either the Hewlett-Packard LaserJet II or III printers, or on any printer for which the PostScript language has been implemented.

troff Input

troff processes each given *file*, or the standard input if none is specified, and prints the formatted result on the standard output. The input must consist of text with formatting commands embedded within it.

troff provides a full suite of commands that set line length, page length and page offset, generate vertical and horizontal motions, indentation, fill and adjust output lines, and center text. The great flexibility of **troff** lies in its acceptance of user-defined macros to control almost all higher-level formatting. For example, the formation of paragraphs, header and footer areas, and footnotes must all be implemented by the user via macros.

troff uses a supeerset of the commands and syntax used by **nroff**, the other COHERENT text-formatter: files prepared for the latter usually can be processed through the former without requiring any changes. **troff** differs from **nroff** in that **nroff** can perform only monospaced formatting, whereas **troff** can handle multiple fonts of type, both monospaced and proportionally spaced. It lets you load font-width tables dynamically, so you can use whatever fonts you have loaded into your printer at a given time. **troff** also lets you move about the page in increments other than sixths of an inch vertically or tenths of an inch horizontally.

troff produces output either in the Hewlett-Packard Printer Control Language (PCL) or PostScript, whichever you prefer. The former can be printed on the Hewlett-Packard LaserJet family of laser printer, and can use any PCL bitmapped "soft font". The latter can be printed on any printer that supports the PostScript language, and can use any font for which you have an Adobe Font Metric (AFM) description. The default is PCL output; to obtain PostScript, use the **-p** command-line option.

Fonts

The COHERENT version of **troff** produces output suitable for printing on a Hewlett-Packard LaserJet or HP-compatible laser printer, using either PCL or PostScript. **troff** knows about three fonts by default, as follows:

\fR	Times Roman
\fI	Times Italic
\fB	Times Bold

troff also recognizes the alternative font names **TR**, **TI** and **TB** for the same three Times family fonts. The default font information for **troff** is in file **/usr/lib/roff/troff_pcl/fonts.r** or **/usr/lib/roff/troff_ps/fonts.r**, as described in detail below.

To use other fonts, you must use the **.lf** request (see below) to load a *font width table*. The font width table is a binary file that describes the width of each character in the font and the printer command (escape sequence) needed to tell the printer to use the font. The program **fwtable** builds a font width table from a PCL bitmap font or from a PostScript AFM description. See its Lexicon entry for details on its use.

troff output includes a printer command for each desired font change. You must use the **hpr** command to download bitmapped *soft fonts* (fonts that are always resident in the printer) to the printer before printing **troff** output that uses the soft fonts. If a specified font

is not available in the printer, the output will not be what you expect.

Command-line Options

Command-line options may be listed in any order. They are as follows:

-d Debug: print each request before execution. This option is very useful when you are writing and debugging new macros.

-D Display the available fonts. These are all the fonts that have been loaded into **troff** with the **.lf** primitive (described below).

-f *name* Write the temporary file into file *name*.

-i *files* Read from the standard input after reading the given *files*.

-k Keep: do not erase the temporary file.

-l Landscape mode: output is rotated 90 degrees, with default size 11 by 8.5 inches rather than 8.5 by 11 inches.

-m*name* Include the macro file **/usr/lib/tmac.***name* in the input stream.

-n*N* Number the first page of output *N*.

-p Produce output for a PostScript printer rather than for a HP-compatible printer.

-r*aN* Set number register *a* to the value *N*.

-r*abN* Set number register *ab* to value *N*. For obvious reasons, *ab* cannot contain a digit.

-x Do not eject to the bottom of the last page when text ends. This option lets you use **troff** interactively, which is especially useful when debugging macros.

If the environmental variable **TROFF** is set when **troff** is invoked, its contents are prefixed to the list of command-line arguments. This allows the user to set commonly used options once in the environment rather than on each **troff** command line.

troff's Primitives

As noted earlier, **troff**'s command set is a superset of that used by **nroff**: see the Lexicon entry on **nroff** for information on the commands and escape sequences shared by **troff** and **nroff**. This article describes the primitives that **troff** does *not* share with **nroff**.

Please note that the basic **troff** unit is one-tenth of a point. A printer's point is 1/12 of a pica, which is in turn one-sixth of an inch; therefore, there are 72 points and 720 **troff** units in an inch.

.co *endmark*
Copy input to output file directly, with no processing. If *endmark* argument is present, **troff** copies input until it finds a line containing *endmark* followed by **\n**. If no *endmark* is given, **troff** copies input until it finds a line containing **.co\n**. This directive is useful for embedding PostScript commands in an input file.

.cs *XX N M*
Set font *XX* to use constant character spacing. The width of each character is *N* divided by 36 ems. If *M* is present, it specifies the width of an em; otherwise, *N* assumes the point size em for the given font.

.fd Display the currently available fonts.

.fp *N XX*
Associate font name *XX* with numeric font position *N*. The given *N* should be a number between 1 and 9. Subsequently, the numeric font position can be used in an escape sequence **\f***N* to select the font. (This nomenclature comes from the days when phototypesetters used print wheels that were set in fixed positions on the device.) The **nroff** primitive **.rf** performs a similar task, and is more flexible in its syntax.

.fz *XX N*
Fix the point size of font *XX* at *N*. The point size of the font will not be affected by subsequent **.ps** commands or **\s***N* point size escapes.

.lf *XX file* [*n*]
Load font width table from *file* and use it for font *XX*. If *file* is not found, **troff** looks for **/usr/lib/roff/troff_pcl/fwt/***file* or **/usr/lib/roff/troff_ps/fwt/***file* (depending on whether the **-p** option is used).

The optional third argument sets the default point size of the loaded font to *n*. Note that this argument takes effect only if **troff** is running in **-p** (PostScript) mode.

For example, to load the font-width table for the PCL bitmapped font **cn090rpn.usp** (which sets Century Roman, nine point, portrait mode) and name it font **RS**, use the command:

```
.lf RS cn090rpn.usp
```

To do the same thing under PostScript, use the command:

```
.lf RS Century_R.fwt 9
```

Thereafter, you can reference font **RS** with either **.ft RS** or **\f(RS**.

Note that the second argument to this primitive must name a font-width table generated by the COHERENT command **fwtable**, not the font itself, although both may have the same name. Look in directories **/usr/lib/roff/troff*/fwt** for the set of font-width tables that are included with COHERENT. If you purchase additional PCL fonts, you must use **fwtable** to generate font-width tables for them. Note, too, that if you are using **troff** in PCL mode, you must both load the font-width table into **troff** and use the command **hpr** to load the font itself into your printer: doing one without the other will not produce the results you desire.

Finally, please note that **.lf** is unique to the COHERENT implementation of **troff**, and cannot be ported to other implementations.

.ps *N* Set point size to *N*. The default point size is 10 point.

.rb *file* Read input from *file* and copy it to the output without processing. This directive is useful for including files containing PostScript routines in the output.

.ss *N* Set the minimum word spacing to *N* divided by 36 ems.

.vs *N***p** Set the vertical spacing to *N* points. The default vertical spacing for **troff** is 11 points.

Escape Sequences

troff recognizes the following escape sequences, in addition to those recognized by **nroff**:

\s'N' Set the point-size escape sequence to *N*. Like the **.ps** primitive, it changes the point size to *N*. The specified *N* may have a leading plus or minus sign to make the new size relative to the current point size.

\X*NN* Output character *NN* where *NN* are two hexadecimal digits. This is useful for forcing **troff** to print characters outside the normal printable range, e.g., those with the high-order bit set. Note that this escape sequence is unique to the COHERENT implementation of **troff** and cannot be ported to other implementations.

Special Characters

troff includes a set of escape sequences for setting special characters. These escape sequences are defined in the files **/usr/lib/roff/troff_*/specials.r**. If you have additional fonts or an extended PostScript cartridge on your printer, you can modify these files to change the current definitions or add new ones.

The following shows the escape sequences currently defined in **specials.r**, and the character each prints:

\(em	—	\(hy	-	\(bu	•	\(sq	[]
\(ru	_	\(14	1/4	\(12	1/2	\(34	3/4
\(fi	fi	\(fl	fl	\(ff	ff	\(Fi	ffi
\(Fl	ffl	\(de	°	\(dg	†	\(fm	'
\(ct	¢	\(rg	®	\(co	©	\(tm	™
\(pl	+	\(mi	–	\(eq	=	\(**	*
\(sc	§	\(aa	´	\(ga	`	\(ul	_
\(sl	/	\(*a	α	\(*b	β	\(*g	γ
\(*d	δ	\(*e	ε	\(*z	ζ	\(*y	η
\(*h	θ	\(*i	ι	\(*k	κ	\(*l	λ
\(*m	μ	\(*n	ν	\(*c	ξ	\(*o	o
\(*p	π	\(*r	ρ	\(*s	σ	\(ts	ς
\(*t	τ	\(*u	υ	\(*f	φ	\(*x	χ
\(*q	ψ	\(*w	ω	\(*A	A	\(*B	B
\(*G	Γ	\(*D	Δ	\(*E	E	\(*Z	Z
\(*Y	H	\(*H	Θ	\(*I	I	\(*K	K
\(*L	Λ	\(*M	M	\(*N	N	\(*C	Ξ
\(*O	O	\(*P	Π	\(*R	P	\(*S	Σ
\(*T	T	\(*U	Y	\(*F	Φ	\(*X	X
\(*Q	Ψ	\(*W	Ω	\(sr	√	\(rn	¯
\(>=	≥	\(<=	≤	\(==	≡	\(~=	≈
\(ap	~	\(!=	≠	\(->	→	\(<-	←
\(ua	↑	\(da	↓	\(mu	×	\(di	/
\(+-	±	\(cu	∪	\(ca	∩	\(sb	⊂
\(sp	⊃	\(ib	⊆	\(ip	⊇	\(in	∈
\(pd	∂	\(gr	∇	\(no	¬	\(is	∫
\(pt	∝	\(es	∅	\(mo	∈	\(br	|
\(dd	‡	\(rh	+	\(lh	<-	\(or	|
\(ci	O	\(lt	/	\(lb	\	\(rt	\
\(rb	/	\(lk	{	\(rk	}	\(bv	|
\(lf	⌊	\(rf	⌋	\(lc	⌈	\(rc	⌉

Example

The following example prints an enormous 'E' on a Hewlett-Packard LaserJet III:

```
.sp |8i
.ps 500
.ce
E
```

Printer Configuration

troff reads several files in directory **/usr/lib/roff/troff_pcl** (for normal **troff**) or **/usr/lib/roff/troff_ps** (for PostScript **troff**) to find printer-specific information. It reads special character definitions from file **specials.r**. It reads font loading requests from file **fonts.r**. It copies file **.pre** at the beginning of the output. It copies file **.post** at the end of the output. In landscape mode, **troff** looks for files **.pre_land** and **.post_land** instead. You can change these files as desired to include printer-specific commands in **troff** output.

Files

/tmp/rof* — Temporary files
/usr/lib/tmac.* — Standard macro packages
/usr/lib/roff/troff_pcl/ — Support files directory for PCL
/usr/lib/roff/troff_ps/ — Support files directory for PostScript
/usr/lib/roff/troff_*/.pre — Output prefix
/usr/lib/roff/troff_*/.pre_land — Output prefix, landscape mode
/usr/lib/roff/troff_*/.post — Output suffix
/usr/lib/roff/troff_*/.post_land — Output suffix, landscape mode
/usr/lib/roff/troff_*/fonts.r — Font definitions
/usr/lib/roff/troff_*/fwt/ — Directory for font width tables
/usr/lib/roff/troff_*/specials.r — Special character definitions

See Also

col, commands, deroff, fwtable, hpr, man, ms, nroff
nroff, The Text-Formatting Language, tutorial

Adobe Systems Incorporated: *PostScript Language Reference Manual*. Reading, Mass.: Addison-Wesley Publishing Company, Inc., 1988.

Adobe Systems Incorporated: *PostScript Language Tutorial and Cookbook*. Reading, Mass.: Addison-Wesley Publishing Company, Inc., 1988.

Emerson, S.L., Paulsell, K.: *troff Typesetting for Unix Systems*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1987 (ISBN 0-13-930959-4).

Lawson, A.: *Printing Types: An Introduction*. Boston: Beacon Press, 1971. *An excellent, one-volume introduction to type and typesetting.*

Tufte, E.W.: *The Visual Display of Quantitative Information*. Cheshire, Conn.: Graphics Press, 1983. *Superbly introduces the subject of graphic design. Especially useful if you wish to explore PostScript.*

University of Chicago Press: *A Manual of Style for Authors, Editors, and Copywriters*, ed. 12, revised. Chicago: University of Chicago Press, 1969. *Still the best one-volume reference for copy editors: if you're going to publish manuals, you should do it right.*

Notes

Like **nroff**, **troff** should be used with the macro packages **ms**, which is found in the file **/usr/lib/tmac.s**, and **man**, which is found in the file **/usr/lib/tmac.an**.

troff output, unlike that of **nroff**, cannot be processed through a terminal driver.

Laser printers cannot print on an area near each edge of the output page. Output sent to the unprintable area will disappear. On some printers, the *logical page* does not correspond to the *physical page*, so printed **troff** output may be offset from the specified position on the physical page.

true — Command

Unconditional success
true

true does nothing, successfully. It always returns zero (i.e., true).

true is useful in shell scripts when you want to execute a condition indefinitely. For example, the following example

```
while true; do
     date
done
```

prints the current date and time on your screen forever (or at least until interrupted by typing **<ctrl-C>**).

See Also

commands, false, ksh, sh

Notes

Under the Korn shell, **true** is an alias for the partial-comment **:**.

tsort — Command

Topological sort
tsort [*file*]

tsort performs a topological sort of a set of input items. The input *file* (or the standard input, if no *file* is given) specifies an ordering on pairs of items. It consists of pairs of items separated by blanks, tabs or newlines. If a pair contains the same item twice, it simply indicates that the item is in the input set. Otherwise, the pair indicates that the first item precedes the second in the ordering.

tsort prints a sorted list of the input items on the standard output.

See Also

commands, sort

Diagnostics

tsort prints an error message on the standard error if its input contains an odd number of items or if the specified ordering includes a cycle.

tty — Command

Print the user's terminal name
tty

tty prints the name of the character-special file that manages your terminal.

Diagnostics

tty prints the message "Not a tty." if the user is not associated with any controlling terminal.

See Also

commands, who

tty.h — Header File

Define flags used with tty processing
#include <sys/tty.h>

tty.h defines flags that are used by routines that handle ttys.

See Also

header files, tty

ttyname() — General Function (libc)

Identify a terminal
char *ttyname(*fd***)**
int *fd***;**

Given a file descriptor *fd* attached to a terminal, **ttyname** returns the complete pathname of the special file (normally found in the directory **/dev**).

Files

/dev/* — Terminal special files
/etc/ttys — Login terminals

See Also

general functions, ioctl(), isatty(), tty(), ttyslot()

Diagnostics

ttyname returns NULL if it cannot find a special file corresponding to *fd*.

Notes

The string returned by **ttyname** kept in a static area, and is overwritten by each subsequent call.

ttys — File Format

gets single user before changing

Describe terminal ports

The file **/etc/ttys** describes the terminals in the COHERENT system. The process **init** reads this file when it brings up the system in multi-user mode.

/etc/ttys contains one line for each terminal. Each line consists of the following four fields:

1. The first field is one character long, and indicates if the device is enabled for logins: '0' indicates that the device is not enabled, and '1' (one) indicates that logins are enabled for the device.

2. The second field is one character long, and indicates whether the device is local (i.e., a terminal) or remote (i.e., a modem): 'r' indicates remote, and 'l' (lower-case **L**) indicates local. If 'r' is used and a password is included for **remacc** (remote access) in **/etc/passwd**, then persons logging in on this device will be required to supply the remote-access password. (See the Lexicon entry for **passwd** for more about **remacc**).

3. The third field is one character long, and sets the baud rate for the device. Note that a device can have either a fixed baud rate, or a variable baud rate. The following table gives the codes for fixed baud rates:

C	110
G	300

I	1200
L	2400
N	4800
P	9600
Q	19200

The common variable-speed codes terminal types are as follows:

0	300, 1200, 150, 110
3	2400, 1200, 300

When a user dials into a variable-speed line, a message is sent to the terminal using the first speed listed. If the message is unintelligible, the user hits the **<break>** key and the system tries the next speed; and so on, until the correct speed is selected.

4. The fourth field names the port that this device is plugged into. The following table names the ports that COHERENT recognizes:

console	The tube and keyboard on your computer
com1l	Serial port **com1**, local device
com1r	Serial port **com1**, remote device
com1pl	Serial port **com1**, local device
com1pr	Serial port **com1**, remote device
com2l	Serial port **com2**, local device
com2r	Serial port **com2**, remote device
com2pl	Serial port **com2**, local device
com2pr	Serial port **com2**, remote device
com3l	Serial port **com3**, local device
com3r	Serial port **com3**, remote device
com3pl	Serial port **com3**, local device
com3pr	Serial port **com3**, remote device
com4l	Serial port **com4**, local device
com4r	Serial port **com4**, remote device
com4pl	Serial port **com4**, local device
com4pr	Serial port **com4**, remote device

Note that if field 2 (described above) says that this is a local device, then you must use a port descriptor that ends in 'l'; likewise, if field 2 states that this is a remote device, the port descriptor must end in 'r'. Doing otherwise will result in trouble. See Lexicon entry **com** for further details.

Do not leave trailing spaces at the end of an entry in **/etc/ttys**. Leaving blanks at the end of a line usually results in errors that state that a device could not be found.

After you have edited **/etc/ttys**, the following command forces COHERENT to re-read the file and use the new descriptions:

```
kill quit 1
```

BSD

Examples

Consider the following **ttys** entry:

```
1lPconsole
```

Field 1 is the first character. Here it is set to '1' (one), which indicates that the device is enabled for logins. Field 2 is the second character. Here it is set to 'l' (lower-case **L**), which indicates that this is a local device. Field 3 is the third character. Here, it is set to 'P', which indicates that the device operates at the fixed baud rate of 9600 baud. This field is

ignored by the console device driver since the console is not a serial device. Finally, field 4 is the remainder of the line. Here, it indicates that the device in question is the console.

Now, consider another example:

```
1r3com3r
```

Field 1 is the first character. Here it is set to '1' (one), which indicates that the device is enabled for logins. Field 2 is the second character. Here it is set to 'r', which indicates that this is a remote device, i.e., a modem. Field 3 is the third character. Here, it is set to '3', which indicates that the device operates at variable baud rates of 2400, 1200, and 300. By hitting the **<break>** key on the terminal, the user can select from among those three baud rates, in that order. Finally, field 4 is the remainder of the line. Here, it indicates that the device in question is plugged into port **com3**, and is accessed via special file **/dev/com3r**.

Files

/etc/ttys

See Also

com, file formats, getty, init, login, stty, terminal, tty

ttyslot() — General Function (libc)

Return a terminal's line number
int ttyslot()

ttyslot returns the number of the line in the file **/etc/ttys** that describes the controlling terminal (see **ttys**).

Files

/dev/* — Terminal special files
/etc/ttys — Login terminals

See Also

general functions, ioctl(), isatty(), tty, ttyname()

Diagnostics

ttyslot returns zero if an error occurs.

ttystat — Command

Get terminal status
/etc/ttystat [-d] *port*

ttystat checks the status of the specified asynchronous *port* in directory **/dev**. It normally just returns an exit status that indicates the status of the *port*. The option **-d** tells **ttystat** to print the status of the *port* on the standard output.

Only the superuser **root** can execute **ttystat**.

Example

The following example prints the status of port **/dev/com2r**:

```
/etc/ttystat -d com2
```

If **/dev/com2r** is enabled, **ttystat** prints:

```
com2r is enabled
```

ttystat finds the port status from the **/etc/ttys** file.

Files

/etc/ttys — Terminal characteristics file

See Also

commands, disable, enable, ttys

Diagnostics

ttystat returns one if the *port* is enabled and zero if the *port* is disabled. It returns -2 if an error occurs.

type checking — Technical Information

Every expression has a *type*, such as **int**, **char**, or **double**. C is not strongly typed, which means that it allows different types to be mixed relatively freely, and be changed (or **cast**) from one type to another.

COHERENT checks types more strictly than the C standard implies. COHERENT's type checking can be enabled or disabled in degrees, using **-VSTRICT** and other "variant" options with the **cc** command.

See Also

cc, technical information, type promotion

typedef — C Keyword

Define a new data type

typedef is a C facility that lets you define new data types. Such definitions are always made in terms of existing data types; for example,

```
typedef long time_t;
```

establishes the data type **time_t**, and defines it to be equivalent to a **long**. By convention, programmer-defined data types are written in capital letters.

Judicious use of the **typedef** facility can make programs easier to maintain, and improve their portability.

See Also

C keyword, manifest constants, portability, storage class

type promotion — Technical Information

In arithmetic expressions, COHERENT promotes one signed type to another signed type by sign extension, and promotes one unsigned type to another unsigned type by zero padding. For example, **char** promotes to **int** by sign extension, whereas **unsigned char** promotes to **unsigned int** by zero padding.

See Also

data formats, technical information

types.h — Header File

Declare system-specific data types
#include <sys/types.h>

The header file **types.h** declares a number of data types that are used throughout the COHERENT system.

See Also

header files

typeset — Command

Set/list variables and their attributes
typeset
typeset [+-]fr
typeset [irx] *variable=value*

The command **typeset** is built into the Korn shell **ksh**. It sets or lists all variables and their attributes.

When called with an argument of the form *variable=value*, it sets variable *variable* to *value*. The following options modify *variable* or *value*:

- **i** Store *value* as an integer
- **r** Make *variable read-only*
- **x** Export *variable* to the environment

When called without an argument, **typeset** lists all variables and their attributes. When called with one of the following options, it lists the variables of the appropriate type. When prefixed with a hyphen '-', it prints the variable plus its value; when prefixed with a plus sign '+', it prints the variable alone:

- **f** List functions instead of variables
- **r** List read-only variables

See Also

commands, ksh

typo — Command

Detect possible typographical and spelling errors
typo [-nrs][*file* ...]

typo proofreads an English-language document for typographical errors. It conducts a statistical test of letter digrams and trigrams in each input word against digram and trigram frequencies throughout the entire document. From this test, **typo** computes an index of peculiarity for each word in the document. A high index indicates a word less like other words in the document than does a low index. Built-in frequency tables ensure reasonable results even for relatively short documents.

typo reads each input *file* (or the standard input if none), and removes punctuation and non-alphabetic characters to produce a list of the words in the document. To reduce the volume of the output, **typo** compares each word against a small dictionary of technical words and discards it if found. The output consists of a list of unique non-dictionary words with associated index of peculiarity, most peculiar first. An index higher than ten indicates that the word almost certainly occurs only once in the document.

typo recognizes the following arguments:

- **-n** Inhibit use of the built-in English digram and trigram statistics, and inhibit dictionary screening of words. More words will be output and the indices of peculiarity will be less useful for short documents.
- **-r** Inhibit the default stripping of **nroff** escape sequences. Normally, **typo** strips lines beginning with '.' and removes the **nroff** escape sequences '\'.

-s Produce output files **digrams** and **trigrams** that contain, respectively, the digram and trigram frequency statistics for the given document. No indices of peculiarity are calculated or printed. If desired, these files may be installed in directory **/usr/dict**.

Files

/tmp/typo* — Intermediate files
/usr/dict/dict — Limited dictionary
/usr/dict/digrams — Digram frequency statistics
/usr/dict/trigrams — Trigram frequency statistics

See Also

commands, nroff, sort, spell

U

umask() — COHERENT System Call

Set file-creation mask
int umask(*mask***)**
int *mask***;**

umask allows a process to restrict the mode of files it creates. Commands that create files should specify the maximum reasonable mode. A parent (e.g. the shell **sh**) usually calls **umask** to restrict access to files created by subsequent commands.

mask should be constructed from any of the permission bits found in **chmod** (the low-order nine bits). When a file is created with **creat** or **mknod**, every bit set in the *mask* is zeroed in *mode*; thus, bits set in *mask* specify permissions that will be denied.

umask returns the old value of the file-creation mask.

Example

Setting *mask* to octal 022 (i.e., 000 010 010) causes a file created with mode octal 0666 to actually have permissions of

```
rw- r-- r--
```

Setting *mask* to zero (i.e., 000 000 000) causes a file created with mode octal 0666 to actually have permissions of

```
rw- rw- rw-
```

See Also

COHERENT system calls, creat(), mknod(), sh

Notes

A file's default permission cannot be set to execute, regardless of the value of *umask*.

umask — Command

Set the file-creation mask
umask [*mask*]

The *file-creation mask* modifies the default mode assigned to each file upon creation. The mode sets the permissions granted by the file's owner, plus other important information about a file.

The command **umask** sets the default file-creation mask to *mask*. *mask* is usually entered as an octal number prefixed by a zero digit. If invoked without an argument, **umask** prints the current file-creation mask in octal.

Note that zero bits in *mask* correspond to permitted permission bits in the target, and that execute permission cannot be enabled via any setting of *mask*. See the Lexicon entries for **umask()** and **chmod** for further details on file mode. The shell executes **umask** directly.

Example

Setting *mask* to octal 022 (i.e., 000 010 010) causes a file created with mode octal 0666 to actually have permissions of

```
rw- r-- r--
```

Setting *mask* to zero (i.e., 000 000 000) causes a file created with mode octal 0666 to actually have permissions of

```
rw- rw- rw-
```

See Also

chmod, commands, ksh, sh, umask()

umount() — COHERENT System Call (libc)

Unmount a file system
umount(*filesystem*)
char **filesystem*;

umount is the COHERENT system call that unmounts a file system. *filesystem* names the block-special file through which the file system is accessed. Note that this must have been previously mounted by a call to **mount**, or the call will fail.

See Also

COHERENT system calls, mount()

umount — Command

Unmount file system
/etc/umount *special*

umount unmounts a file system *special* that was previously mounted with the **mount** command.

The script **/bin/umount** calls **/etc/umount**, and provides convenient abbreviations for commonly used devices. For example, typing

```
umount f0
```

executes the command

```
/etc/umount /dev/fha0
```

The system administrator should edit this script to reflect the devices used on your specific system.

Files

/etc/mtab — Mount table
/dev/*
/bin/umount — Script that calls **/etc/umount**

See Also

clri, commands, fsck, icheck, mount

Diagnostics

Errors can occur if *special* does not exist or is not a mounted file system.

unalias — Command

Remove an alias
unalias *alias* ...

The command **unalias** is built into the Korn shell **ksh**. It removes each *alias*.

See Also

alias, commands, ksh

uncompress — Command

Uncompress a compressed file
uncompress [**-w** *tmpfile*] [*file* ...]

uncompress uncompressses one or more *files* that had been compressed by the command **compress**.

Each *file*'s name must have the suffix **.Z**, which was appended onto it by **compress**; otherwise, **uncompress** prints an error message and exits. When **uncompress** has uncompressed a *file*, it removes the **.Z** suffix from that file's name.

If no *file* is specified on the command line, **uncompress** uncompresses matter read from the standard input, and writes its output to the standard output.

Older versions of **uncompress** could only uncompress files that had been compressed with option **-b12** or lower, with **-b12** being the default. The edition of **uncompress** released with COHERENT version 3.1 now handles values up to 16 by using RAM device **/dev/ram1** for temporary storage. For this reason, it is strongly advised that you not use **/dev/ram1** as a RAM disk.

The **-w** option allows the user to specify an alternate temporary storage file to **uncompress**. The default value for *tempfile* when the **-w** option is omitted is **/dev/ram1**.

See Also

commands, compress, ram, zcat

ungetc() — STDIO (libc)

Return character to input stream
#include <stdio.h>
int ungetc (*c*, *fp*) **int** *c*; **FILE** **fp*;

ungetc returns the character *c* to the stream *fp*. *c* can then be read by a subsequent call to **getc**, **gets**, **getw**, **scanf**, or **fread**. No more than one character can be pushed back into any stream at once. A call to **fseek** will nullify the effects of a previous **ungetc**.

Example

For an example of this function, see **fgetc**.

See Also

fgetc(), getc(), STDIO

Diagnostics

ungetc normally returns *c*. It returns **EOF** if the character cannot be pushed back.

union — C Keyword

Multiply declare a variable

A **union** defines an area of storage that can accept any one of several types of data. In effect, it is a multiple declaration of a variable. For example, a **union** may be declared to consist of an **int**, a **double**, and a **char** *. Any one of these three elements can be held by the **union** at a time, and will be handled appropriately by it. For example, the declaration

```
union {
     int number;
     double bignumber;
     char *stringptr;
} example;
```

allows **example** to hold either an **int**, a **double**, or a pointer to a **char**, whichever is needed at the time. All of these have the same address. The elements of a **union** are accessed like those of a **struct**: for example, to access **number** from the above example, type **example.number**.

unions are helpful in dealing with heterogeneous data, especially within structures; however, you are responsible for keeping track of what data type the **union** is holding at any given time. Passing a **double** to a **union** and then reading the **union** as though it held an **int** will yield results that are unpredictable, and probably unwelcome.

Example

For an example of how to use a **union** in a program, see the entry for **byte ordering**.

See Also

C keywords, initialization, struct, structure

uniq — Command

Remove/count repeated lines in a sorted file
uniq [-cdu] [-*n*] [+*n*] [*infile*[*outfile*]]

uniq reads input line by line from *infile* and writes all non-duplicated lines to *outfile*. The input file must be sorted. **uniq** uses the standard input or output if either *infile* or *outfile* is omitted. The following describes the available options:

-c Print each line once, discarding duplicate lines; before each line, print the number of times it appears within the file.

-d Print only lines that are duplicated within the file; print each line only once; do not print any counts.

-u Print only lines that are *not* duplicated within the file.

uniq by default behaves as if both **-u** and **-d** were specified, so it prints each unique line once.

Optional specifiers allow **uniq** to skip leading portions of the input lines when comparing for uniqueness.

-n Skip *n* fields of each input line, where a field is any number of non-white space characters surrounded by any number of white space characters (blank or tab).

+n Skip *n* characters in each input line, after skipping fields as above.

See Also

comm, commands, sort

units — Command

Convert measurements
units [-u]

units is an interactive program that tells you how to convert one unit of measurement into another. It prompts you for two quantities with the same dimensions (e.g., two

measurements of weight, or two of size). It first prints the prompt "You have:" to ask for the unit you wish to convert from, and then prints the prompt "You want:" for the unit you wish to convert to.

Example

The following example returns the formula for convert fortnights into days:

```
You have: fortnight
You want: days
* 14
/ 0.071428
```

The following fundamental units are recognized: **meter**, **gram**, **second**, **coulomb**, **radian**, **bit**, **unitedstatesdollar**, **sheet**, **candle**, **kelvin**, and **copperpiece** (shillings and pence).

A quantity consists of an optional number (default 1) and a dimension (default none). Numbers are floating point with optional sign, decimal part and exponent. Dimensions may be specified by fundamental or derived units, with optional orders. A quantity is evaluated left to right: a factor preceded by a '/' is a divisor, otherwise it is a multiplier. For example, the earth's gravitational acceleration may be entered as any of the following:

```
9.8e+0 m+1 sec-2
32 ft/sec/sec
32 ft/sec+2
```

British equivalents of US units are prefixed with **br**, e.g. **brpint**. Some other units include **c** (speed of light), **G** (gravitational constant), **R** (gas law constant), **phi** (golden ratio), % (1/100), **k** (1 024), and **buck** (United States dollar).

/usr/lib/units is an ASCII file that contains conversion tables. The binary file **/usr/lib/binunits** may be recreated by using the **-u** option.

See Also

bc, **commands**, **conv**

Files

/usr/lib/units — Known units
/usr/lib/binunits — Binary encoding of units file

Diagnostics

If the ASCII file **/usr/lib/units** has been changed more recently that the binary file **/usr/lib/binunits**, **units** prints a message and regenerates the binary file before continuing; this takes up to a few minutes, depending on the speed of your system.

The error message "conformability" means that the quantities are not dimensionally compatible. For example, **m/sec** and **psi**. **units** prints each quantity and its dimensions in fundamental units.

Notes

There are the inevitable name collisions: **g** for gram vs. **gee** for Earth's gravitational acceleration, **exp** for the base of natural logarithms vs. **e** for the charge of an electron, **ms** for (plural) meters vs. **millisecond**, and of course **batman** for the Persian measure of weight rather than the Turkish.

unlink() — COHERENT System Call (libc)

Remove a file
int unlink(*name*) **char** **name*;

unlink removes the directory entry for the given file *name*, which in effect erases *name* from the disk. *name* cannot be opened once it has been **unlink**ed. If *name* is the last link, **unlink** frees the i-node and data blocks. Deallocation is delayed if the file is open. Other links to the file remain intact.

Example

This example removes the files named on the command line.

```
main(argc, argv)
int argc; char *argv[];
{
	int i;

	for (i = 1; i < argc; i++) {
		if (unlink(argv[i]) == -1) {
			printf("Cannot unlink \"%s\"\n", argv[i]);
			exit(1);
		}
	}
	exit(0);
}
```

See Also

COHERENT system calls, link(), ln, rm, rmdir

Diagnostics

unlink returns zero when successful. It returns -1 if *file* does not exist, if the user does not have write and search permission in the directory containing *file*, or if *file* is a directory and the invoker is not the superuser.

unmkfs — Command

Construct a prototype file system
unmkfs [-*prefix*] *directory nblocks* [*file*]

unmkfs scans *directory* and builds prototype files with which you can build file systems on backup disks.

If *prefix* is given, it creates files *prefix*.**p01**, *prefix*.**p02**, etc. If it is not given, **unmkfs** writes its output to the standard-output device.

nblocks gives the maximum size of a prototype file. COHERENT current defines a block as being 512 bytes (half a kilobyte); thus, to make the maximum size of a prototype file 10 kilobytes, set *nblocks* to 20.

The *file* option tells **unmkfs** to suppress all files in *directory* that are older than *file*. If it is not used, then **unmkfs** builds prototypes for all files in *directory*.

unmkfs provides a useful way to back up file systems onto floppy disks. To do this, perform the following steps:

1. **unmkfs** a directory, producing prototype files.
2. Format one floppy disk for each prototype file.
3. Using the prototype files in sucession, **mkfs** each floppy disk. This puts the indicated files onto floppy disk, preserving links.

Later, you can use the command **cpdir** to restore all the files from the floppy disks, or you can use **cp** to restore individual files.

See Also

commands, mkfs

Notes

unmkfs builds a file system in memory as it does its work. With large directory structures, it can run out of memory.

unsigned — C Keyword

Data type

unsigned tells the compiler to treat the variable as an unsigned value. In effect, this doubles the largest absolute value that that type can hold, and changes the lowest storage value to zero.

See Also

C keywords, data type

until — Command

Execute commands repeatedly
until *sequence1* [**do** *sequence2*] **done**

The shell's **until** loop executes the commands in *sequence1*. If the exit status is nonzero, the shell then executes the commands in the optional *sequence2* and repeats the process until the exit status of *sequence1* is zero. Because the shell recognizes a reserved word only as the unquoted first word of a command, both **do** and **done** must occur either unquoted at the start of a line or preceded by ';'.

The shell commands **break** and **continue** may be used to alter control flow within an **until** loop. The contruct **while** has the same form as **until** but the sense of the test is reversed.

The shell executes **until** directly.

See Also

break, commands, continue, ksh, sh, test, while

update — System Maintenance

Update file systems periodically
/etc/update

update periodically calls **sync** to write to the disk all file system data that are in memory. It never exits.

The initialization command file **/etc/rc** normally executes **update**. It should not be executed directly.

See Also

init, sync, system maintenance

uproc.h — Header File

Definitions used with user processes
#include <sys/uproc.h>

uproc.h defines constants and structures used by routines that manage user processes.

See Also

header files

USER — Environmental Variable

Name user's identifier
USER=*user_identifier*

The environmental variable **USER** names your login identifier. For example, if your login identifier is **fwb**, then by typing **set** you will see the entry **USER=fwb**. **USER** is set by **login**.

See Also

environmental variables, ksh, login, sh

ustar — Command

disks formatted with fdformat -av /dev/rfd0d

Process tape archives
ustar -c[vw] [-f *archive***]** *file* ...
ustar -r[vw] [-f *archive***]** *file* ...
ustar -t[v] [-f *archive***]**
ustar -x[lmovw] [-f *archive***]** [*file* ...]

ustar reads and writes archive files that conform to the Archive/Interchange File Format specified in IEEE Standard 1003.1-1988.

Options

ustar recognizes the following command-line options:

-c Create a new archive. Write each *file* into the newly created archive

-r Append each *file* to the end of the archive.

-t Display a directory (table of contents) of the archive.

-x Extract each *file* from the archive. If *file* matches a directory whose contents had been written onto the archive, extract that directory recursively. If *file* does not exist on the system, **ustar** creates it with the same mode as the one in the archive, except that the set-user-id and get-group-id modes are not set unless you have appropriate privileges.

If *file* exists, **ustar** does not change file modes except as described above. It restores the owner, the group, and the modification time if possible.

If the command line does not name a *file*, **ustar** extracts the entire contents of the archive. Note that if several files with the same name are in the archive, the last one overwrites all earlier ones.

-f Use the next argument on the command line as the name of the archive instead of the default, which is standard input or standard output, whichever is appropriate for the options given. Thus, you can use **ustar** in a pipe. If you specify '-' as the archive name with option **-f**, **ustar** uses the default input and output streams.

-l Order **ustar** to report if it cannot resolve all of the links to the files being archived. If **-l** is not specified, no error messages are written to the standard output. This modifier is valid only with the **-c** or **-r** options.

-m Do not restore the modification times; set the modification time of the file to the time of extraction. This modifier is invalid with the **-t** option.

-o Give extracted files your user and group identifiers, rather than those on the archive. This modifier is only valid with the **-x** option.

-v Verbose operation. With this option, **ustar** prints the name of each file it processes, preceded by the option letter. With the **-t** option, the **-v** option tells **ustar** to give more information about the archive's entries than just their names.

-w Print the action to be taken and the name of the file, then wait for your confirmation. If you type a word beginning with 'y', **ustar** performs the action. Any other input means "no". This modifier is invalid with the **-t** option.

See Also

commands, cpio, dd, find, pax

Notes

ustar was developed by Mark H. Colburn and sponsored by The USENIX Association. Copyright © 1989 by Mark H. Colburn (mark@jhereg.MN.ORG). All rights reserved. See the compressed **tar** archive **/usr/src/alien/pax.tar.Z** for full descriptions of copyright, restrictions, and licensing terms.

ustar is provided in binary form per the licensing terms set forth by the author. It is distributed as a service to COHERENT customers, as is. It is not supported by Mark Williams Company. *Caveat utilitor.*

utime() — COHERENT System Call

Change file access and modification times

#include <sys/types.h>
int utime(*file, times***)**
char ****file*;**
time_t *times***[2];**

utime sets the access and modification times associated with the given *file* to times obtained from *times*[0] and *times*[1], respectively. The time of last change to the attributes is set to the time of the **utime** call.

This call must be made by the owner of *file* or by the superuser.

Files

<sys/types.h>

See Also

COHERENT system calls, restor, stat()

Diagnostics

utime returns -1 on errors, such as if *file* does not exist or the invoker not the owner.

utmp.h — Header File

Login accounting information
#include <utmp.h>

/etc/utmp contains a **utmp** entry for every user currently logged into the COHERENT system. The structure **utmp** is defined in the the header file **utmp.h**, as follows:

```
#define DIRSIZ  14
struct utmp {
        char    ut_line[8];     /* terminal name */
        char    ut_name[DIRSIZ];/* user name */
        time_t  ut_time;        /* time of login */
};
```

If either the user name or terminal name is cleared, the entry is unused. The element **ut_line** is the name of the special file for the user's terminal, and is normally found in the directory **/dev**. **ut_time** gives the date and time the user logged into COHERENT.

The file **/usr/adm/wtmp** maintains a record of all logins and logouts, and may be summarized by the command **ac**. The processes **login** and **init** write entries into the file **wtmp**; neither creates the file, so login accounting is disabled unless **/usr/adm/wtmp** exists.

Entries in **wtmp** are identical to those in **utmp**. A null string in the **ut_name** field indicates a logout. The following three special terminal names may be found in **wtmp**. When the system is booted, **init** writes a **ut_line** entry of '~'. When the time is changed with the command **date**, it writes an entry giving the old date ('|') and an entry giving the new date ('}'). This allows **ac** to adjust connect times appropriately.

Files

<utmp.h>
/etc/utmp
/usr/adm/wtmp

See Also

ac, date, file formats, header files, init, login, who

utsname.h — Header File

Define utsname structure
#include <sys/utsname.h>

utsname.h defines the structure **utsname**. This structure holds information that describes a given release of the COHERENT system.

See Also

header files

uucheck — Command

Sanity check the UUCP system
uucheck [-fsv]

uucheck is a script which calls a series of programs designed to locate and fix problems in the UUCP system. The phases of the **uucheck** system all accept the same arguments.

uucheck recognizes the following options:

-f Attempt to fix errors in the UUCP system. Note that it is not possible to automatically correct all errors. Only **root** (the super user) can use this option.

-s Run in "silent" mode (i.e., generate no output). Normally **uucheck** will report all errors and warnings encountered. With this option set, only errors internal to **uucheck** will be reported. Option **-s** overrides option **-v**.

-v Generate verbose output messages. This will include messages on what is being checked, and often longer messages suggesting remedies.

The phases of **uucheck** are:

uucheckperms Check file permissions.
uucheckname Check **/etc/uucpname**.
uucheckclock Look for lock files.

Notes

This system does not and probably cannot identify all possible failure modes of the UUCP system — nothing beats an experienced UUCP administrator.

Files

/usr/lib/uucheck/ — Directory for phases

See Also

commands, UUCP

uucico — Command

uucico -x3 -Ssite2 -l

Transmit data to or from a remote site
/usr/lib/uucp/uucico [-r1] [-s*site* **] [-S***site* **] [-x***level* **]**

uucico is the UUCP command that actually transfers files to or from a remote *site*. Its syntax is as follows:

-r0 Act as slave in polling process; that is, carry out the orders of another **uucico** that has dialed into your system.

-r1 Act as master in polling process; that is, dial out to another system and give it orders. This is the default.

-s*site*
Name *site* as a place to be polled. *site* must name one of the entries in **/usr/lib/uucp/L.sys**.

-S*site*
Name *site* as a place to be polled. *site* must be a site described in file **L.sys**. Unlike the **-s** option, this option forces transfer of any queued files even if the time is not one specified as valid in **L.sys**.

-x*level*
Set the debugging level, where *level* is a number between one and ten, inclusive. **uucico** prints all messages at or below the current debugging level. The following gives the class of messages controlled by each level of debugging:

0 No additional logging
1 The call and pre-protocol negotiation
2 Conversation level
3 File transfer
4 Spool files
5 Messages sent out during call/pre-protocol

6 High-level protocol
7 Medium-level protocol
8 Low-level protocol (framing and such)
9 Actual packet data
10 Reading configuration files

If you do not use this option, **uucico** sets the debugging level to zero. Please note that enabling the debugging option causes certain non-COHERENT versions of **uucico** to fail.

The messages sent by **uucico** are differentiated by the first letter of the message.

Example

To poll the site *sys* at five minutes after the hour, each hour, put the following entry into **/usr/lib/crontab:**

```
05 * * * * /usr/lib/uucp/uucico -ssys -rl
```

Files

/usr/lib/uucp/L.sys — List of reachable systems
/usr/spool/uucp/.Log/uucico/*sitename* — **uucico** activities log file for *sitename*
/usr/spool/uucp/*sitename* — Spool directory for work

See Also

commands, cron, uucp, UUCP, uulog, uutouch, uuxqt

UUCP — Overview

Unattended communication with remote systems

UUCP stands for "UNIX to UNIX copy". It is a system of commands that allows you to exchange files with other COHERENT or UNIX systems, in an unattended manner. With UUCP, you can send mail to other systems, upload files, and execute commands. When configured correctly, UUCP also lets other users upload files to your system, copy files from it, and execute commands. All this can be done without your having to sit at your console and type commands; thus, files can be transferred in the small hours, when telephone rates are lower and computers are relative free.

UUCP gives you access to the Usenet, a nation-wide network of UNIX and COHERENT users. Access to the Usenet will let you exchange mail with any of the thousands of Usenet users, receive mail from them, download source code for many useful programs, and read the latest news on a host of subjects. For details on contacting UUNET, a commercially accessible Usenet site, enter the command:

```
phone uunet
```

The UUCP protocol is implemented through a suite of commands and files. The following Lexicon entries relate to UUCP:

L-dev	File that describes devices used by UUCP
L.sys	File that describes systems contacted by UUCP
Permissions	File that sets remote system permissions
uucheck	Sanity check the UUCP system
uucico	Transmit data to a remote site
uucp	Prepare files for transmission
uucpname	File that sets your system's UUCP name
uudecode	Decode a binary file sent from a remote system
uuencode	Encode a binary file for sending to a remote system
uuinstall	Install UUCP on your system
uulog	Read UUCP log files
uumvlog	Move UUCP log files to backup archive
uuname	List UUCP names of known systems
uurmlock	Remote UUCP lock files
uutouch	Touch a file to trigger **uucico** poll
uux	Execute command on a remote system
uuxq	Execute commands requested by a remote system

The following sections discuss problems that can arise when using UUCP, as diagnosed by the Mark Williams Technical Support Staff.

Using Trailblazer Modems With UUCP

The Trailblazer modem has been designed to be used with UUCP. It is extremely fast and extremely accurate; however, some users reported problems in using COHERENT UUCP with a Trailblazer modem. The following describes how Mark Williams Company has configured the Trailblazer modem that it uses under COHERENT at 9600 baud.

To begin, the following gives the permissions in **/usr/lib/uucp** for selected files:

```
-rw-r--r--  1 uucp     uucp       196 Wed May 23 10:16 L-devices
-rw-r--r--  1 uucp     uucp       740 Tue May 22 15:40 L.sys
-rw-r--r--  1 uucp     uucp      2151 Wed Jul 11 10:44 Permissions
```

The following gives partial contents of file **/usr/lib/uucp/L-devices**:

```
#type    line    remote  baud    brand
#-----  ------  ------  ----    -----
ACU     com3l   com3r   2400    tb2400
ACU     com3l   com3r   9600    tbfast
```

The following gives partial contents of file **/usr/lib/uucp/L.sys** (note that **#** should be replaced by the actual phone number):

```
uunet Any ACU 9600 # FAST \d\r\c in:-\r-in: \dmwc\r\c rd: PASSWORD\r
```

The following gives partial contents of **/usr/lib/uucp/Permissions**:

```
MACHINE=uunet LOGNAME=uuunet \
     COMMANDS=rmail:rnews: \
     READ=/usr/spool/uucppublic:/tmp \
     WRITE=/usr/spool/uucppublic:/tmp \
     SENDFILES=yes REQUEST=no
```

The following gives permissions and partial contents of file **/etc/ttys**:

```
-rw-r--r--  1 root   root   163 Wed Jul 11 11:36 /etc/ttys

1lPconsole
1rPcom3r
```

Finally, the following gives permissions on **/dev/com3r** (while enabled):

```
c--s------  1 root   root   21  38 Wed Jul 11 11:50 /dev/com3r
```

To use the Trailblazer, log in as **root**, and type the following commands:

```
disable com3r
kermit cbl 9600 /dev/com3l > /tmp/modem_dump
```

Note that we disabled the "remote" device but used the "local" modem device when using **kermit**. This allows us to access the modem registers without having to wait for the modem to assert the carrier detect signal.

While talking to the Trailblazer, we sent it **ATN?** followed by the **<Enter>** key. We then entered ^ (the circumflex) followed by the letter **C** in order to exit from **kermit**. The following gives the results contained in file **/tmp/modem_dump**:

```
kermit: connected...

E0 F1 M1 Q9 T V0 W0 X3 Y0 &P0 &T4     Version BA5.01
S00=001 S01=000 S02=043 S03=013 S04=010 S05=008 S06=002 S07:060 \
     S08=002 S09=006
S10=007 S11=070 S12=050 S18=000 S25=005 S38=000
S41=000 S45=000 S47=004 S48:001 S49=000
S50=000 S51:252 S52=000         S54:003 S55=000 S56=017 S57=019 \
     S58:000 S59=000
S60=000 S61:230 S62=003 S63=001 S64=000 S65=000 S66=000 S67=000 \
     S68=255 S69=000
S90=000 S91=000 S92=000 S94=001 S95=000 S96=001
S100=000 S101=000 S102=000 S104=000 S105=001
S110=255 S111=255 S112=001
S121=000 S130=002 S131:001 S255=000
N0:
N1:
N2:
N3:
N4:
N5:
N6:
N7:
N8:
N9:
0
kermit: disconnected.
```

/etc/ttys Problems

Sometimes, UUCP problems will arise because the entry in file **/etc/ttys** for the serial port your modem is using, is either missing or is incorrect.

To discover which port UUCP thinks your modem is using, invoke the command **uuinstall**. Then, under its **Devices** option, look at the *remote* line. The remote device you specified on this line must be described in file **/etc/ttys**. If it is not, then you will find statements in the log file for the site you are calling stating that a device was not found.

If you see errors in the log files that state that a device is not found, and you have checked the "remote" entry in **uuinstall** against the contents of **/etc/ttys**, the next possible cause may be the **/etc/ttys** entry for the port. Look for trailing spaces at the end of the line that describes the **com** port. If you find a space at the end of a line, delete it.

com Port Driver Permissions

By far, the most common problem deals with permissions associated with the **com** port devices. If you are trying to get UUCP to call out on a port, and keep seeing errors of the form

```
Dial failed, Line Problem
```

in the output of **uulog**, you may need to fix permissions on the port in question.

If you are using one **com** port both for remote logins and to call out using UUCP, note the following: When a port is enabled for remote access via a modem, the permissions for the port are changed so that only the superuser **root** can access the port. This prevents someone from inadvertently trying to send data out the port. When the port is later disabled so that UUCP can dial out, the permissions for that **com** port *are not* changed to give everyone access to the port. Remember, when UUCP is executing, it is just another user with the name "uucp" and does not have **root** privileges.

To set the permissions properly, use the command **chmod** to reset the permission of the **com** port device in directory **/dev**. For example, if your UUCP connection is via **com1r**, log in as the superuser **root** and enter the command:

```
chmod 666 /dev/com1*
```

Usually, serial ports should have read and write permissions turned on for all users. The main exception to this rule is that a port enabled for logins becomes readable and writeable only by **root**. This not a problem for UUCP as the port is disabled (and permissions expanded) temporarily when **uucico** runs.

Lock Files and Temporary Files

UUCP controls access to the modem and to various directories and sites via a set of "lock files". This is to prevent UUCP from tripping over its own feet by attempting to write more than one file to the same site at the same site.

When a UUCP session fails, it may fail to remove all of its lock files before it exits, depending upon the seriousness of the failure. "Stale" lock files and temporary files in directory will prevent UUCP from accessing a given site or even from working altogether. Symptoms of this problem are messages in the log files that state:

```
Site locked
```

This, of course, is not indicative of a problem unless no UUCP connection has been made recently (within the last minute or so). To cure this problem, log in as the superuser **root** and then enter the command:

```
uurmlock
```

This will remove all "stale" lock or temporary files.

UUCP Configuration Files

By far the most common cause of problems are mistakes in one or another UUCP configuration file. If problems persist, check all UUCP configuration files against the descriptions found in correspondingly named Lexicon articles. The files in question are **ttys**, **L-devices**, **L.sys**, and **Permissions**.

UUCP Executable File Permissions

UUCP commands can invoke each other from time to time. If a UUCP executable file's permissions are set incorrectly, that command may be prevented from being executed under certain conditions, or from reading or writing certain key files.

Key UUCP executable files are **/usr/lib/uucp/uucico**, **/usr/lib/uucp/uuxqt**, **/usr/bin/uucp**, and **/usr/bin/uux**. These files must belong to user and group **uucp**. Permissions on these files must be 6511 (-r-s--s--x). See the Lexicon article for the command **chmod** for further details on how to reset permissions for files.

UUCP Connects, but ...

Once UUCP is dialing out, it is extremely difficult to diagnose problems, as they can occur at either end of the connection. In most cases, one must know both systems to diagnose problems related to communication problems. Check the following:

- Check your chat scripts. Contact the other system's system administrator to be sure that you are expecting the correct prompts in the chat script for the system you are calling.
- Use the debugging mode of **uucico** to watch communications. Debugging mode is accessed using the **uucico** command suffixed by a **-x#**, where **#** is 1 to 9 which determines the debugging level. For example

  ```
  /usr/lib/uucp/uucico -Smwcbbs -x1
  ```

  Please note, the **uucico** debugging option is incompatible with certain non-COHERENT versions of **uucico**.
- Note also that the COHERENT default mailer does not yet support domains. If domain information is being sent to your system, an error may result and possibly cause unexpected results or errors from **uucico**. The error will usually state that a log file could not be written to. Alternate mailers are available from the various COHERENT archive sites, as well as the MWC UUCP bulletin board system.

Remote Won't Accept Files ... Where'd My Mail Go?

If you see messages in your log files that a site would not accept a file from your site, the other site may not have its permissions set to allow you to send files to it, or to write the files you are sending to a directory that you specified.

When sending files across systems, check the length of the site name. Currently, COHERENT can only work with a seven-character or shorter site name. If you are using an eight-character or longer site name, COHERENT will not properly distribute files transferred from your site. To change the name of your system, edit the contents of file **/etc/uucpname**; and to change the name of your system's domain, edit the contents of file **/etc/domain**.

See Also

com, commands, L-dev, L.sys, Permissions, terminal, uucico, uucp, uucpname, uudecode, uuencode, uuinstall, uulog, uumvlog, uuname, uurmlock, uutouch, uux, uuxqt

UUCP, Remote Communications Utility, tutorial

uucp — Command

Ready files for transmission to other systems
uucp [**-cCdfmr**] [**-g***grade*] [**-n***user*] [**-s** *dir*] [**-x***n*] *source* ... *dest*

uucp copies files *source1* through *sourceN* to the destination system *dest*. Either source or destination files can contain specifications for the remote system.

uucp recognizes the following options:

-c Do not copy the source file into spool directory. This is the default.

-C Copy the source file into spool directory.

-d Create directories as required on the destination system.

-f Do not make intermediate directories for the file copy.

-g*grade*
Assign a grade (a single ASCII character) to indicate the importance of the files being transmitted: the lower the value of *grade*, the more important the file.

-m Send mail to requester when the file is sent.

-n*user*
Notify *user* on destination system when the file is received. Note that *user* may contain a path, e.g.:

```
-nuser!site
```

-r Spool transfer request, but do not initiate **uucico**.

-x*n* Assign debug level *n* (0 to 9).

Examples

The first example copies file **foo** to directory **/bar** on system **george**:

```
uucp foo george!/bar
```

The next example copies file **/foo** from system **george** into directory **/tmp** on your system:

```
uucp george!/foo /tmp
```

The next example copies file **/foo** from system **george** into file or directory **/bar** on system **ivan**:

```
uucp george!/foo ivan!/bar
```

Note that this assumes your system can talk to both **george** and **ivan** and that your system has permission to read file **/foo** on system **george** as well as to write file **/bar** on system **ivan**.

The next example downloads files **/foo** and **/bar** from remote systems **ivan** and **george** into directory **/tmp** on your system:

```
uucp ivan!/foo george!/bar /tmp
```

Files

/usr/lib/uucp/L.sys — List of reachable systems
/usr/lib/uucp/Permissions — List of system permissions
/usr/spool/uucp/.Log/*/*sitename* — **uucp** activities log files for *sitename*
/usr/spool/uucp/*sitename* — Spool directory for work

See Also

commands, mail, uucico, UUCP, uudecode, uuencode, uutouch, uuwatch, uuxqt

uucpname — System Maintenance

Set the system's UUCP name
/etc/uucpname

The file **/etc/uucpname** sets the name by which your system is known to all other system with which it communicates via UUCP. To rename your system, simply change the contents of this file. Note that only the superuser **root** can change edit this file.

See Also

system maintenance, UUCP

uudecode — Command

Decode a binary file sent from a remote system
uudecode [*file*]

uudecode takes a **file** encoded by **uuencode** and translates it back to binary. Any leading and trailing lines added by **uucp** are discarded.

If the *file* is not specified, standard input is read.

Example

Consider the file **tmp** consisting of:

```
begin 644 sys
M5&AE('%U:6-K(&)R;W=N(&9O>"!J=6UP<R!O=F5R('1H92!L87IY(&1O9RX*

end
```

Note that the third line is a space followed by a newline. To decode it, type:

```
uudecode tmp
```

The output contained in file **sys** will be:

The quick brown fox jumps over the lazy dog.

See Also

commands, uucp, UUCP, uuencode

Notes

The user on the remote system must be able to write the file.

uuencode — Command

Encode a binary file for transmission
uuencode [*source*] *remotedest*

uuencode prepares a binary file for transmission to a remote destination via **uucp**. **uuencode** takes binary input and produces an encoded version, consisting of printable ASCII characters, on standard output, which may be redirected or piped to **uucp**. If *source* is not specified, the standard input is read.

The format of the encoded file is as follows:

1. A *header* line starting with the characters **begin** followed by a space. This is followed by the mode of the file in octal (see **chmod** for details) and the name of the output file specified on the command line. These last two fields are also separated by a space. The mode and the system name can be changed by directing the output into a file and editing it.

2. The *body* of the file, consisting of a number of lines, each no more than 62 characters long, including a newline character. Each line starts with a character count written as a single ASCII character, representing an integer value from 0 (octal 40) to 63 (octal 135) giving the number of characters in the rest of the line. This is followed by the encoded characters and a newline. The last line of the body is a line consisting of an ASCII space (octal 40).

3. The trailer line has just the characters **end** on a line by itself.

The encoding is done by taking three bytes and storing them in four characters, six bits per character.

Example

To encode the file **tmp** consisting of the line

```
The quick brown fox jumps over the lazy dog.
```

to be sent to the remote system **george**, enter:

```
uuencode tmp george
```

The output will be:

```
begin 644 george
M5&AE('%U:6-K(&)R;W=N(&9O>"!J=6UP<R!O=F5R('1H92!L87IY(&1O9RX*

end
```

Note that the third line consists of a space followed by a newline.

See Also

commands, uucp, UUCP, uudecode

Notes

The file is expanded by more than one third, causing increased transmission time. This can be a factor when sending large files.

uuinstall — Command

Install UUCP
uuinstall

uuinstall assists with the installation of UUCP. It uses screen templates, help lines, and prompts to help walk you through the installation of devices, remote systems, site names, domains, and permissions. For a detailed description of its use, see the tutorial on UUCP in the front of this manual.

See Also

commands, UUCP

Notes

Only the superuser **root** can execute **uuinstall**.

uulog — Command

Examine UUCP operations
uulog [**-fx**] [*system*]

uulog copies the last part of the file **/usr/spool/uucp/.Log/uucico/***system* to see what **uucico** has done recently. *system* names the remote system whose logfile will be examined. If it is not specified, logfiles for all systems are displayed.

uulog recognizes the following options:

-f Similar to the command **tail -f**: this forces **uulog** to display UUCP activity as it is written into the log file, until you interrupt it by typing **<ctrl-C>**.

-x Display the log files for the command **uuxqt** rather than **uucico**.

Files

/usr/spool/uucp/.Log/uucico/*system*— **uucico** log file for *system*
/usr/spool/uucp/.Log/uuxqt/*system*— **uuxqt** log file for *system*

See Also

commands, uucico, uucp, UUCP, uuxqt

uumvlog — Command

Archive UUCP log files
uumvlog *days*

uumvlog copies all UUCP log files into backup files, named for their respective commands and the date upon which the backup was performed. *days* gives the number of days for which backup files should be kept: if a backup file is more than *days* days old, then **uumvlog** will delete it.

This command should be run by **cron**, because the UUCP log files can threaten to exhaust available file space on a small system unless they are chopped back daily. For directions on how to do this, see the tutorial for UUCP or the Lexicon entry for **cron**.

Files

/usr/spool/uucp/.Log/*command/system*— UUCP log files

See Also

commands, crontab, uucico, uucp, UUCP, uuxqt

uuname — Command

List UUCP names of known systems
uuname [**-l**]

uuname lists the names of all systems reachable directly by **uucp**. When used with the **-l** option, it reads and prints the contents of file **/etc/uucpname**, which holds the name of your local system.

Files

/etc/uucpname — Name of local system
/usr/lib/uucp/L.sys — Site and remote login data list

See Also

commands, uucico, uucp, UUCP, uulog

uurmlock — Command

Remove UUCP lock files
uurmlock

UUCP uses a system of lock files to ensure that sites are polled in an orderly manner. It creates a lock file named after the site being polled, to prevent more than one invocation of **uucico** or another UUCP command from polling the same site at the same time. On occasion, if UUCP fails or crashes, it will neglect to clean up its lock files, thus preventing itself from polling the locked sites.

The command **uurmlock** removes all UUCP lock files. You should run this if you suspect that UUCP has died in a disorderly manner and has left lock files lying around unattended.

Before you run **uurmlock**, examine the output of the command **ps** to ensure that no UUCP command is running at the moment (and so has legitimately locked a site). Note that only the superuser **root** can run **uurmlock**.

Files

/usr/spool/uucp/LCK.*— UUCP lock files

See Also

commands, UUCP

uutouch — Command

Touch a file to trigger uucico poll
uutouch *system*

uutouch creates an empty control file for *system* in the directory **/usr/spool/uucp/***system*. This forces UUCP to poll *system* when **uucico** is called with the option **-sany**. If the empty file for *system* aready exists, it is left alone.

There are three types of files in the spool directory **/usr/spool/uucp/***system*:

C. Command file.

D. Data file.

X. Execute file.

Example

A typical usage is to put the following line into **/usr/lib/crontab:**

```
0 7 * * * /usr/lib/uucp/uutouch george
```

This forces UUCP to schedule a poll to the remote system **george** at 7 AM local time. The actual poll take place when **uucico** is started.

Files

/usr/spool/uucp/*sitename* — Directory for uucp work files

See Also

commands, cron, uucico, uucp, UUCP, uuxqt

uux — Command

Execute a command on a remote system
uux [**-a** *user*] [**-g** *grade*] [**-rnpz**] *command-string*

The command **uux** executes commands on a remote system. **uux** works in conjunction with the UUCP system. It is not generally used by the end user, but is instead called by the various UUCP subsystem components to request that work be performed at a remote system. For security reasons, you can execute on the remote system only the commands that are explicitly permitted by the remote system, as described in the entry for your system in the remote system's copy of **/usr/lib/uucp/Permissions**.

If all permissions are in order, an appropriately named **X.** file is created in the remote system's directory **/usr/spool/uucp/***yoursystem*, where *yoursystem* gives the name by which the remote system knows your system. This file is then executed by the remote system's copy of **uuxqt**.

command-string consists of a command name followed by zero or more arguments. Both the command name and the arguments may be prefixed by an optional system name (sitename) and an exclamation mark. Note that all special characters must be escaped or enclosed in quotation marks to avoid being processed by your system's shell.

For example, the simplest form of the **uux** command is:

```
uux host!command arg0 arg2 argN
```

where *host* is the name of the remote system being contacted, as described in the file **/usr/lib/uucp/L.sys**, *command* is the name of the command to execute on the remote system, and *arg0* through *argN* give the arguments to *command*.

If an argument names a file, that file can either be on the remote system, on your system, or on some third system. For example, the command

```
uux widget!lpr /usr/sally/herfile
```

requests site **widget** to print its own file **/usr/sally/herfile**. On the other hand, the command

```
uux widget!lpr !$HOME/myfile
```

requests that site **widget** print on its line printer the file **myfile** from your home directory on your home system. Note that the '!' that prefixes **myfile** indictes that it is on your system. Finally, the command

```
uux widget!lpr ZEUS!/usr/fred/hisfile
```

requests that **widget** print file **/usr/fred/hisfile** which resides on site **ZEUS**. Note that if **widget** does not know how to contact site **ZEUS**, the command will fail.

If you wish, you can embed the shell operators '<', '>', ';', or '|' within a **uux** command. This lets you construct a more powerful command than you could do otherwise. Commands that contain these operators must, of course, be quoted so ensure that your shell does not interpret them. For example, the command

```
uux "widget!pr /usr/sally/herfile > ZEUS!~/fred/hisfile"
```

tells **uux** to use **pr** to format its file **/usr/sally/herfile**, and write the output into file **/usr/spool/uucppublic/fred/hisfile** on site **ZEUS**. (Note that the tilde '~', as always, is a synonym for **/usr/spool/uucppublic**.) Again, the command will fail if you do not have appropriate permissions on **widget** or if **widget** does not have appropriate permissions on

ZEUS.

The operator '-' lets you use the standard input when constructing a **uux** command. For example, the command

```
who | uux - widget!lpr
```

executes the **who** command on your system, pipes the output to **uux**, and tells **uux** to invoke the command **lpr** on remote system **widget** to print the list of users on your system.

uux will attempt to transfer any needed input files to the system which will be executing the requested command. You must enclose in parentheses any output files generated by *command*, to distinguish them from input file names.

uux recognizes the following options:

-a *user*
: Use *user* as the name of the requester. The default is the requester's login name.

-g *grade*
: *grade* is a single ASCII character that indicates the importance of the files being transmitted: the lower the value of *grade*, the more important the files.

-r Queue up the **uux** request but do not invoke **uucico** to actually handle the transfer. The default is to initiate **uucico**.

-n Suppress notification of command completion. The default is to send mail to the requester after the command has been run.

-p Input to **uux** will be via a pipe or input redirection.

-z Notify requester when *command-string* succeeds. The default is to not generate a notification.

Examples

The following script uses a remote system to print files. Print files specified on the command line are sent unprocessed to system **prnsrvr** for printing using command **lpr**. Note that since the **-r** option is specified to **uux**, **uucico** will not be invoked automatically, thus causing the requests simply to be queued.

```
for i in $*
do
     uux -r prnsrvr!lpr !$i
done
```

The next example copies file **/foo** from system **george** and file **/bar** from system **norms** to your system and then invokes command **cmp** to compare the contents of the files. The results of the comparison are placed in output file **/tmp/cmp.results** on the local system, and notification of command completion is sent via electronic mail.

```
uux -z "!cmp -l george!/foo norms!/bar >/tmp/cmp.results"
```

Note that this example assumes that your system can talk to both **george** and **norms** and that your system has permission to read file **/foo** on system **george** as well as to read file **/bar** on system **norms**.

The last example runs a remote C compile on system **cserver** using local file **mycode.c** as input and producing executable file **mycode** as output. Any C compiler error messages will be placed in file **/tmp/errors** on the local system.

```
uux 'cserver!cc -O -o (!mycode) !mycode.c >!/tmp/errors'
```

See Also

commands, UUCP, uuxqt

uuxqt — Command

Execute commands requested by a remote system
uuxqt

uuxqt takes the execute files, those marked with the prefix **X.** in the directory **/usr/spool/uucp/***sitename*, and executes them. It will only execute programs for which the remote system has permission.

uuxqt may be called by either **uucp** or **uucico.** It is not generally considered a user-callable program.

Files

/usr/spool/uucp/*sitename* — Directory for execute files

See Also

commands, uucico, uucp, UUCP, uux

V

v7sgtty.h — Header File

UNIX Version 7-style terminal I/O
#include <v7sgtty.h>

v7sgtty.h defines structures and constants used by routines that perform terminal I/O in the manner of UNIX version 7.

See Also

header files, sgtty.h, tty.h

va_arg() — Variable Arguments

Return pointer to next argument in argument list
#include <stdarg.h>
typename ***va_arg**(*listptr*, *typename*)
va_list *listptr*, *typename*;

va_arg returns a pointer to the next argument in an argument list. It can be used with functions that take a variable number of arguments, such as **printf** or **scanf**, to help write such functions portably. It is always used with **va_end** and **va_start** within a function that takes a variable number of arguments.

listptr is of type **va_list**, which is defined in the header **stdarg.h**. This object must first be initialized by the macro **va_start**.

typename is the name of the type for which **va_arg** is to return a pointer. For example, if you wish **va_arg** to return a pointer to an integer, *typename* should be of type **int**.

va_arg can only handle "standard" data types, i.e., those data types that can be transformed to pointers by appending an asterisk '*'.

Example

For an example of this macro, see the entry for **variable arguments**.

See Also

variable arguments

Notes

If there is no next argument for **va_arg** to handle, or if *typename* is incorrect, then the behavior of **va_arg** is undefined.

The ANSI Standard demands that **va_arg** be implemented only as a macro. If its macro definition is suppressed within a program, its behavior is undefined.

va_end() — Variable Arguments

Tidy up after traversal of argument list
#include <stdarg.h>
void va_end(*listptr*)
va_list *listptr*;

va_end helps to tidy up a function after it has traversed the argument list for a function that takes a variable number of arguments. It can be used with functions that take a variable number of arguments, such as **printf** or **scanf**, to help write such functions portably. It should be used with the routines **va_arg** and **va_start** from within a function that takes a variable number of arguments.

listptr is of type **va_list**, which is declared in header **stdarg.h**. *listptr* must first have been initialized by macro **va_start**.

Example

For an example of this function, see the entry for **variable arguments**.

See Also

variable arguments

Notes

If **va_list** is not initialized by **va_start**, or if **va_end** is not called before a function with variable arguments exits, then the behavior of **va_end** is undefined.

va_start() — Variable Arguments

Point to beginning of argument list
#include <stdargs.h>
void va_start(*listptr, rightparm*)
va_list *listptr, type rightparm*;

va_start is a macro that points to the beginning of a list of arguments. It can be used with functions that take a variable number of arguments, such as **printf** or **scanf**, to help implement them portably. It is always used with **va_arg** and **va_end** from within a function that takes a variable number of arguments.

listptr is of type **va_list**, which is a type defined in the header **stdarg.h**.

rightparm is the rightmost parameter defined in the function's parameter list. Its type is set by the function that is using **va_start**. Undefined behavior results if any of the following conditions apply to **rightparm**: if it has storage class **register**; if it has a function type or an array type; or if its type is not compatible with the type that results from the default argument promotions.

Example

For an example of this macro, see the entry for **variable arguments**.

See Also

variable arguments

Notes

The ANSI Standard demands that **va_start** be implemented only as a macro. If the macro definition of **va_start** is suppressed within a program, the behavior is undefined.

variable arguments — Overview

The ANSI Standard mandates the creation of a set of routines to help implement functions, such as **printf** and **scanf**, that take a variable number of arguments. These routines are called from within another function to help it handle its arguments.

These routines are declared or defined in the header **stdarg.h**, and are as follows:

va_arg()	Return pointer to next argument in argument list
va_end()	Tidy up after an argument list has been traversed
va_start()	Initialize object that holds function arguments

va_arg and **va_start** must be implemented as macros; **va_end** must be implemented as a library function. All three use the special type **va_list**, which is an object that holds the arguments to the function being implemented.

Example

The following example concatenates multiple strings into a common allocated string and returns the string's address.

```
#include <stdarg.h>
#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>

char *
multcat(numargs)
int numargs;
{
     va_list argptr;
     char *result;
     int i, siz;

     /* get size required */
     va_start(argptr, numargs);
     for(siz = i = 0; i < numargs; i++)
          siz += strlen(va_arg(argptr, char *));

     if ((result = calloc(siz + 1, 1)) == NULL) {
          fprintf(stderr, "Out of space\n");
          exit(EXIT_FAILURE);
     }
     va_end(argptr);

     va_start(argptr, numargs);
     for(i = 0; i < numargs; i++)
          strcat(result, va_arg(argptr, char *));
     va_end(argptr);
     return(result);
}

int
main(void)
{
     printf(multcat(5, "One ", "two ", "three ",
          "testing", ".\n"));
}
```

See Also

libraries, stdarg.h

vi — Command

Clone of Berkeley-style screen editor

vi [*options*] [*+cmd*] [*file1 ... file27*]

vi is a link to the editor **elvis**, which is a clone of the UNIX editors **ex** and **vi**. For details on how to run **vi**, see the entry for **elvis** in the Lexicon.

See Also

commands, ed, ex, elvis, me, view

Notes

elvis is copyright © 1990 by Steve Kirkendall, and was written by Steve Kirkendall (kirkenda@cs.pdx.edu or ...uunet!tektronix!psueea!eecs!kirkenda), assisted by numerous volunteers. It is freely redistributable, subject to the restrictions noted in included documentation. Source code for **elvis** is available through the Mark Williams bulletin board, USENET, and numerous other outlets.

elvis is distributed as a service to COHERENT customers, as is. It is not supported by Mark Williams Company. *Caveat utilitor.*

view — Command

Screen-oriented viewing utility
view *file1 ... file27*

view is a link to **elvis**, which is a clone of the UNIX **vi/ex** set of editors. Invoking **elvis** through this link forces it to operate solely in read-only mode, just as the UNIX **view** utility operates.

For information on how to use this version of **view**, see the Lexicon page for **elvis**.

See Also

commands, ed, elvis, ex, me, vi

Notes

view is copyright © 1990 by Steve Kirkendall, and was written by Steve Kirkendall (kirkenda@cs.pdx.edu or ...uunet!tektronix!psueea!eecs!kirkenda), assisted by numerous volunteers. It is freely redistributable, subject to the restrictions noted in included documentation. Source code for **virec** is available through the Mark Williams bulletin board, USENET, and numerous other outlets.

elvis is distributed as a service to COHERENT customers, as is. It is not supported by Mark Williams Company. *Caveat utilitor.*

virec — Command

Recover the modified version of a file after a crash
virec [-d *tmpdir*] *textfilename...*
virec </tmp/elv*XXX*

virec extracts the most recent version of a text file from a temporary file in **/tmp**.

When you edit a file with **elvis**, only about five kilobytes of the file are stored in RAM; the rest is stored in a file in **/tmp**. **virec** extracts the "undo" version from the file stored in **/tmp**. This is most useful when the system (or **elvis**) crashes in the middle of a long editing session, since the "undo" version of the file contains everything except your last change.

There are two ways to use **virec**. The first, and most common, way to invoke **virec** is to give it the name of the file you were editing; it finds the matching file in **/tmp** and writes the newest available version of the file over the existing version. It then deletes the **/tmp** file.

The second way is to use the '<' to let **virec** read a particular **/tmp** file via **stdin**. Use this method when you either have forgotten which file you were editing and want to see its contents, or when you wish to recover a file without losing either the **/tmp** file or the current version of the text file.

The **-d** option tells **virec** to look for a temporary file in *directory* rather than in **/tmp**.

Files

/tmp/elv* — Temporary file created by **elvis**

See Also

commands, elvis

Notes

virec is copyright © 1990 by Steve Kirkendall, and was written by Steve Kirkendall (kirkenda@cs.pdx.edu or ...uunet!tektronix!psueea!eecs!kirkenda), assisted by numerous volunteers. It is freely redistributable, subject to the restrictions noted in included documentation. Source code for **virec** is available through the Mark Williams bulletin board, USENET, and numerous other outlets.

Please note that this program is distributed as a service to COHERENT users, but it is not supported by Mark Williams Company. *Caveat utilitor.*

void — C Keyword

Data type

The keyword **void** indicates that the function does not return a value. Using **void** declarations makes programs clearer and is useful in error checking. For example, a function that prints an error message and calls **exit** to terminate a program should be declared **void** because it never returns. A function that performs a calculation and stores its result in a global variable (rather than **return**ing the result), or one that returns no value, should also be declared **void** to prevent the accidental use of the function in an expression.

See Also

C keywords

volatile — C Keyword

Qualify an identifier as frequently changing

The type qualifier **volatile** marks an identifier as being frequently changed, either by other portions of the program, by the hardware, by other programs in the execution environment, or by any combination of these. This alerts the translator to re-fetch the given identifier whenever it encounters an expression that includes the identifier. In addition, an object marked as **volatile** must be stored at the point where an assignment to this object takes place.

See Also

C keyword, const

Notes

Although COHERENT recognizes this keyword, the semantics are not implemented in this release. Thus, storage declared to be **volatile** might have references removed by optimizations that the compiler performs. The compiler will generate a warning if the peephole optimizer is enabled and the keyword **volatile** is detected.

W

wait() — COHERENT System Call (libc)

Await completion of a child process
wait(*statp*)
int **statp*;

wait suspends execution of the invoking process until a child process (created with **fork**) terminates. It returns the process identifier of the terminating child process. If there are no children or if an interrupt occurs, it returns -1.

If it is successful, **wait** returns the process identifier of the terminated child process. In addition, **wait** fills in the integer pointed to by *statp* with exit-status information about the completed process. If *statp* is NULL, **wait** discards the exit-status information.

wait fills in the low byte of the status-information word with the termination status of the child process. A child process may have terminated because of a signal, because of an exit call, or have stopped execution during **ptrace**. Termination with **exit**, which is normal completion, gives status 0. Other terminations give signal values as status (as defined in the article on **signal**). The **0200** bit of the status code indicates that a core dump was produced. A status of **0177** indicates that the process is waiting for further action from **ptrace**.

The high byte of the returned status is the low byte of the argument to the **exit** system call.

If a parent process does not remain in existence long enough to **wait** on a child process, the child process is adopted by process 1 (the initialization process).

See Also

_exit(), COHERENT system calls, fork(), ksh, ptrace(), signal(), sh

wait — Command

Await completion of background process
wait [*pid*]

Typing the character '&' after a command tells the shell **sh** to execute it as a *background* (or *detached*) process; otherwise, it is executed as a *foreground* process. You can perform other tasks while a background process is being executed. The shell prints the process id number of each background process when it is invoked. **ps** reports on currently active processes.

The command **wait** tells the shell to suspend execution until the child process with the given *pid* is completed. If no *pid* is given, **wait** suspends execution until all background processes are completed. If the process with the given *pid* is not a child process of the current shell, **wait** returns immediately.

The shell executes **wait** directly.

See Also

commands, ps, sh

Notes

If a subshell invokes a background process and then terminates, **wait** will return immediately rather than waiting for termination of the grandchild process.

wall — Command

Send a message to all logged-in users
/etc/wall

wall types a message to every user currently logged into the COHERENT system, with the exception of the sender. It can be used to inform users of information of general interest; for example, that man has landed on the moon, or that the system is going down in 15 minutes.

wall reads the message to be broadcast from the standard input until end of file. When it sends the message, it prefaces it with the herald "Broadcast message ...", which includes an audible warning. Only the superuser should invoke **/etc/wall** (to override access protections of the target terminals).

Files

/etc/utmp — Current users file
/dev/tty*

See Also

commands, msg, who, write

Diagnostics

The message "Cannot send to *user* on *ttyname*" indicates that **wall** cannot write to the given *user*.

wc — Command

Count words, lines, and characters in text files
wc [-clw] [*file*...]

wc counts words, lines, and characters in each *file*. If no *file* is given, **wc** uses the standard input. If more than one *file* is given, **wc** also prints a total for all of the files.

wc defines a *word* to be a string of characters surrounded by white space (blanks, tabs, or newlines). It defines the number of lines to be the number of newline characters in the file, plus one.

wc recognizes the following options:

-c Count only characters.

-l Count only lines.

-w Count only words.

The default action is to print all counts.

See Also

commands

whence — Command

List a command's type
whence [-v] *command* ...

The command **whence** is built into the Korn shell **ksh**. It lists the type for each *command*. Option **-v** lists function and alias values as well.

See Also

commands, ksh

whereis — Command

Locate source, binary, and manual files
whereis [-bmrsu] [-BMS *dir* ... **-f]** *name* ...

The command **whereis** locates source files, binary files (executables), and manual pages (documentation) that match a given *name*. Prior to searching, **whereis** strips *name* of any path information, extensions, and the **s.** prefix.

By default, **whereis** searches the following directories:

Sources	*Binaries*	*Manual Pages*
/usr/src/cmd	/bin	/usr/man/ALL
/usr/src/games	/usr/bin	/usr/man/COHERENT
/usr/src/local	/usr/games	/usr/man/KERNEL
/usr/src/alien	/usr/local	/usr/man/MULTI
/usr/include	/etc	
/usr/include/sys	/lib	
	/usr/lib	

Options

whereis recognizes the following command-line options:

-b Search only for binary files.

-B Use the directory list specified by *dir* instead of the default directory list for binary files.

-f Terminate the directory list introduced by options **-B**, **-M**, or **-S**, and treat any additional command-line arguments as file names to be searched for.

-m Search only for manual pages (documentation files).

-M Use the directory list specified by *dir* instead of the default directory list for manual pages.

-r Search recursively downward from the directories specified by *dir* or from the default directories. This option is useful when the searched directories contain sub-directories. By default, **whereis** searches only the directories specified or the default directories.

-s Search only for source files.

-S Use the directory list specified by *dir* instead of the default directory list for source files.

-u Search for "unusual" files. A file is said to be unusual if it does not have one entry for each of the three search categories.

Please note that if you use options **-B**, **-S**, or **-M**, you must use the **-f** option to terminate the directory list specified by *dir*.

Example

The following example finds all commands in directory **bin** that have either no corresponding source code in directory **src** or no corresponding documentation in directory **doc**:

```
whereis -u -M doc -S src -B bin -f bin/*
```

See Also

commands, find, qfind, which

Notes

whereis is copyright © 1980,1990 by The Regents of the University of California. All rights reserved.

whereis is distributed as a service to COHERENT customers, as is. It is not supported by Mark Williams Company. *Caveat utilitor.*

which — Command

Locate executable files
which *command* ...

which displays the full path name associated with *command*. It searches the directories named by environment variable **PATH** for the first executable that matches *command* and that you have permission to execute. If **which** can find no executable that matches your request, an error message is displayed.

Example

The following example displays the path names that correspond to commands **write**, **vi**, **myprog**, and **fsck**:

```
which write vi myprog fsck
```

See Also

commands, find, PATH, qfind, whereis

while — C Keyword

Introduce a loop
while(*condition*)

while is a C keyword that introduces a conditional loop. *condition* is tested on reiteration of the loop, and the loop ends when *condition* is no longer satisfied. For example,

```
while (foo < 10)
```

introduces a loop that will continue until the variable **foo** is reset to ten or greater. Note that the statement

```
while (1)
```

will loop forever, unless interrupted by a **break**, **goto**, or **return** statement.

See Also

break, C keywords, continue, do, for

while — Command

Execute commands repeatedly
while *sequence1* [**do** *sequence2*] **done**

The shell construct **while** controls a loop. It first executes the commands in *sequence1*. If the exit status is zero, the shell executes the commands in the optional *sequence2* and repeats the process until the exit status of *sequence1* is nonzero. Because the shell recognizes a reserved word only as the unquoted first word of a command, both **do** and **done** must occur unquoted at the start of a line or preceded by ';'.

The shell commands **break** and **continue** may be used to alter control flow within a **while** loop. The **until** construct has the same form as **while**, but the sense of the test is reversed.

The shell executes **while** directly.

See Also

break, commands, continue, ksh, sh, test, until

who — Command

Print who is logged in
who [*file*] **[am i]**

The command **who** prints the names of the users who are logged in to the system. For each user, **who** prints her name, terminal name, login date, and login time. The form **who am i** prints this information only about yourself.

If *file* is specified, **who** uses it instead of **/etc/utmp** to obtain information about who is logged in. This is useful, for example, with the file **/usr/adm/wtmp**, which contains a continuous record of logins, logouts and reboots. When *file* is specified, **who** displays logouts; otherwise, they are suppressed.

Files

/etc/utmp — To get user information

See Also

ac, commands, sa

wildcards — Definition

Wildcards are characters that, in some circumstances, can represent a range of ASCII characters. Another name for them is "metacharacters". The wildcards available under the COHERENT are as follows:

? Match any one character.

***** Match any number of characters, or no characters at all.

[ ] A set of characters enclosed between '[' and ']' match any one character of the set. Sets of characters may include ranges, such as **[a-z]** for all lower-case letters or **[0-9]** for all numerals.

[!] A set of characters enclosed between '[!' and ']' match any one character *except* one of the set. Sets of characters may include ranges, such as **[a-z]** for all lower-case letters or **[0-9]** for all numerals. For example, the command

```
ls [!abc]*
```

prints the names of all files *except those that begin with* **a**, **b**, *or* **c**.

**** Ignore the special meaning of a wildcard.

See Also

definitions, egrep, patterns, pnmatch()

write() — COHERENT System Call

Write to a file
int write(*fd*, *buffer*, *n*)
int *fd*; **char** **buffer*; **int** *n*;

write writes *n* bytes of data, beginning at address *buffer*, into the file associated with the file descriptor *fd*. Writing begins at the current write position, as set by the last call to either **write** or **lseek**. **write** advances the position of the file pointer by the number of characters written.

Example

For an example of how to use this function, see the entry for **open**.

See Also

COHERENT system calls, STDIO

Diagnostics

write returns -1 if an error occurred before the **write** operation commenced, such as a bad file descriptor *fd* or invalid *buffer* pointer. Otherwise, it returns the number of bytes written. It should be considered an error if this number is not the same as *n*.

Notes

write is a low-level call that passes data directly to COHERENT. It should not be mixed with high-level calls, such as **fread**, **fwrite**, or **fopen**.

write — Command

Converse with another user
write *user* [*tty*]

The COHERENT system provides several commands that allow users to communicate with each other. **write** allows two logged-in users to have an extended, interactive conversation.

write initiates a conversation with *user*. If *tty* is given, **write** looks for the *user* on that terminal; this is useful if a user is marked as being logged in on more than one device. Otherwise, **write** holds the conversation with the first instance of *user* found on any tty.

If found, **write** notifies *user* that you are beginning a conversation with him. All subsequent lines typed into **write** are forwarded to the *user*'s terminal, except lines beginning with '!', which are sent to the shell **sh**. Typing end of file (usually **<ctrl-D>**) terminates **write** and sends *user* the message "EOT" to tell him that communication has ended.

Two users typing lines to **write** at about the same time can cause extreme confusion, so users should agree on a protocol to limit when each is typing. The following protocol is suggested. One user initiates a **write** to another, and waits until the other user replies before beginning. The first user then types until he wishes a reply and suffixes "o" (over) to indicate he is through. The other user does the same, and the conversation alternates until one user wishes to terminate it. This user types "oo" (over and out). The other user replies in the same way, indicating he too is finished. Finally each of the users leave **write** by typing end-of-file (usually **<ctrl-D>**).

Any user may deny others the permission to **write** to his terminal by using the command **mesg**.

Files

/etc/utmp
/dev/*

See Also

commands, mail, mesg, msg, sh, wall, who

Notes

You should use **write** only for extended conversations. Use **msg** to send brief communications to a logged in user, and **mail** to communicate with a user not currently logged in. **wall** broadcasts a message to all logged in users.

wtmp see utmp.h (pg 1012)

X

xgcd() — Multiple-Precision Mathematics

Extended greatest-common-divisor function

#include <mprec.h>
void xgcd(*a*, *b*, *r*, *s*, *g*)
mint **a*, **b*, **r*, **s*, ***g*;

The COHERENT system includes a suite of routines that allow you to perform multiple-precision mathematics. The function **xgcd** is an extended version of the greatest-common-division function. It sets the multiple-precision integer (or **mint**) pointed to by *g* to the greatest common divisor of the **mint** pointed to by *a* and that pointed to by *b*. It also sets the **mints** pointed to by *r* and *s* so the following relation holds:

$$g = a * r + b * s$$

r, *s*, and *g* must all be distinct.

See Also

multiple-precision mathematics

Y

yacc — Command

Parser generator
yacc [*option* ...]*file*
cc y.tab.c [-ly]

Many programs process highly structured input according to given rules. Compilers are a familiar example. Two of the most complicated parts of such programs are *lexical analysis* and *parsing* (sometimes called *syntax analysis*). The COHERENT system includes two powerful tools called **lex** and **yacc** to assist you in performing these tasks. **lex** takes a set of lexical rules and writes a lexical analyzer, whereas **yacc** takes a set of parsing rules and writes a parser; both output C source code that can be compiled into a full program.

The term *yacc* is an acronym for "yet another compiler-compiler". In brief, the **yacc** input *file* describes a context free grammar using a BNF-like syntax. The output is a file **y.tab.c**; it contains the definition of a C function **yyparse**(), which parses the language described in *file*. The output is ready for processing by the C compiler **cc**. Ambiguities in the grammar are reported to the user, but resolved automatically by precedence rules. The user must provide a lexical scanner **yylex**(), which you may generate with **lex**. The **yacc** library includes default definitions of **main**, **yylex**, and **yyerror**, and may be included with the option **-ly** on the **cc** command line.

yacc recognizes the following options:

-d Enable debugging output; implies **-v**.

-hdr *headerfile*
Put the header output in *headerfile* instead of **y.tab.h**.

-l *listfile*
Place a description of the state machine, tokens, parsing actions, and statistics in file *listfile*.

-st Print statistics on the standard output.

-v Verbose option. Like **-l**, but places the listing in file **y.output** by default.

The following options are useful if table overflow messages appear:

-nterms *N*
Allow for *N* nonterminals; default, 100.

-prods *N*
Allow for *N* productions (rules); default, 175.

-states *N*
Allow for *N* states; default, 300.

-terms *N*
Allow for *N* terminal symbols; default 100.

-types *N*
Allow for *N* types; default, ten.

Files

y.tab.c — C source output
y.tab.h — Default C header output
y.output — Default listing output

/lib/yyparse.c — Protoparser
/tmp/y[ao]* — Temporaries
/usr/include/action.h — Header referenced by protoparser
/usr/lib/liby.a — Library

See Also

cc, commands, lex
Introduction to yacc, Yet Another Compiler-Compiler
DeRemer F, Pennello TJ: *Efficient computation of LALR(1) lookahead sets.* SIGPLAN conference, 1979.

Diagnostics

yacc reports the number of R/R (reduce/reduce) and S/R (shift/reduce) conflicts (ambiguities) on the standard error stream.

yes — Command

Print infinitely many responses
yes [*string*]

With no argument, **yes** prints the string **y\n** forever. If a *string* is named on the command line, then **yes** prints it forever.

Example

The following example scribbles the string **foo\n** over a high-density, 5.25-inch floppy disk in drive 0 (drive A):

```
yes foo >/dev/fha0
```

See Also

commands

Z

zcat — Command

Concatenate a compressed file
zcat [*file* ...]

zcat concatenates one or more *files* that had been compressed with the command **compress**. It uncompresses each *file* "on the fly," and prints the uncompressed text onto the standard output.

If no *file* is specified on the command, **zcat** uncompresses matter read from the standard input.

Older versions of **zcat** could only uncompress files that had been compressed with option **-b12** or lower, with **-b12** being the default if the option was omitted. This release of **zcat** now handles values up to **-b16** by using RAM device **/dev/ram1** for temporary storage. For this reason, it is strongly advised that you not use **/dev/ram1** as a RAM disk.

Example

zcat is extremely useful for extracting selected items from large archives; it spares you the overhead of having to uncompress the entire archive just to get at one or two files.

For example, to extract **myfile** from the compressed archive **backup.tar.Z**, use the following command line:

```
zcat backup.tar.Z | tar xf - myfile
```

See Also

commands, compress, ram, uncompress

zerop() — Multiple-Precision Mathematics

Indicate if multi-precision integer is zero
#include <mprec.h>
int zerop(*a***)**
mint ****a*;**

The COHERENT system includes a suite of routines that allow you to perform multiple-precision mathematics. The function **zerop** returns true if the multiple-precision integer (or **mint**) pointed to by *a* is zero; otherwise, it returns false.

See Also

multiple-precision mathematics

Section 17:

Appendix: Error Messages

The following lists the error messages produced by major utilities within COHERENT.

COHERENT System Error Messages

The following gives the error messages returned by the COHERENT kernel. The messages describe two categories of error:

- **Hardware:** These messages indicate serious problems with your system hardware. If any appears, you need to contact a representative of the hardware manufacturer. Note that the symbol '#' in the following messages stands for a number that appears when the kernel prints the message on the console. When reporting the problem, be sure to include the number actually printed out.
- **Halts:** These messages appear when COHERENT has crashed.

When you see a *halt* message on the console, copy it down, as well as all other information on the screen. If the advice offered in this section does not help the problem, call Mark Williams Support.

Arena # too small (hardware)

Bad block # (alloc) (hardware)
: The kernel attempted to allocate a block of memory, only to find that there was something physically wrong with it.

Bad block # (free) (hardware)
: The kernel attempted to free a block of memory, only to find that there was something physically wrong with it.

Bad free # (hardware)

Bad freelist (halt)
: The *freelist* is a list of free blocks on the disk. The COHERENT system maintains this list so it can see where it can write data on the disk. This message indicates that the freelist has been corrupted somehow. To fix this problem, run **/etc/shutdown** to return to single-use mode; use **sync** to flush the buffers; use **umount** to unmount the affected file system; and then run **fsck** to repair the file system.

Bad segment count (hardware)

Bus error at # (hardware)

Cannot allocate stack (hardware)

Cannot create process (hardware)

Corrupt arena (hardware)

Illegal instruction at # (hardware)

Inode # busy (hardware)

Inode table overflow (hardware)

Not a separate I/D machine (hardware)

Out of i-nodes (halt)
A COHERENT file system has one i-node for each file it maintains. The number of i-nodes is set when the file system is created. If you have numerous small files on a file system, it is possible to exhaust that file system's resources even though the command **df** shows that space remains on the file system. To get around this problem, you must delete files, one file for each i-node needed; or you must use **ar** to archive a mass of files. To do this, first use **/etc/shutdown** to return the system to single-user mode, as described above. Delete files, or use **ar** as described above. Then use **sync** to flush all buffers, and use the command **umount** to unmount the affected file system. Then run **fsck** on the affected file system before rebooting. **fsck** checks COHERENT file systems and fixes them if necessary. Consult the Lexicon entry on **fsck** before you use this program for the first time.

Out of space (*m*,*n*) (halt)
When this error message appears, your file system still has i-nodes but the alloted disk space has been exhausted; perhaps you have a few large files that are eating up disk space. To get around this problem, you must delete or compress files to clear up disk space. First, use **/etc/shutdown** to return to single-user mode, as described above; then delete files or compress them as described above until enough space has been cleared to allow you to continue your work. Use **sync** to flush buffer, use **umount** to unmount the affected file system, and run **fsck** on the affected file system. Then reboot.

Random trap (hardware)

Raw I/O from non user (hardware)

System too large (hardware)

Swapio bad parameter (hardware)

Swapio error (hardware)

Compiler Error Messages

The following gives the error messages returned by the COHERENT C compiler, the assembler **as**, and the linker **ld**. The messages are in alphabetical order, and each is marked as to whether it is a *fatal*, *error*, *warning*, or *strict* condition. The compilation phases are **cpp**, the preprocessor; **cc0**, the parser; **cc1**, the code generator; and **cc2**, the optimizer.

A fatal message usually indicates a condition that caused the compiler to terminate execution. Fatal errors from the later phases of compilation often cannot be fixed, and may indicate problems in the compiler.

An error message points to a condition in the source code that the compiler cannot resolve. This almost always occurs when the program does something illegal, e.g., has unbalanced braces.

Warning messages point out code that is compilable, but may produce trouble when the program is executed. A strict message refers to a passage in the code that is unorthodox and may not be portable.

Assembler Error Messages

. (error)
: Dot label error. This indicates that a period was used as a label, e.g., ".:".

a (error)
: Addressing error. This is generated by nearly any kind of operand/instruction mismatch or semantic error in address fields.

string: cannot create (error)
: The assembler cannot create the output file it was requested to create. This often is due to a problem with the output device; check and make sure that it is not full, and that it is working correctly.

internal error, c=*number* in expr. (error)
: The assembler has detected a situation that "should not occur". Please send a copy of the source code that triggered this error to Mark Williams Company. For immediate help during business hours, contact Mark Williams Company technical support.

m (error)
: Multiple definition. The offending line is involved in the multiple definition of a label.

o (error)
: An unrecognized opcode mnemonic was found. Contrast this with error 'q', where the opcode is recognized but the syntax is in error.

p (error)
: Phase error. The value of a label changed during the assembly. An instruction has a size that differs between the first and second passes.

q (error)
: Questionable syntax. The assembler has no idea how to parse this line, and it has given up.

r (error)
: Relocation error. The program attempted to create or use an expression in a way that the linker cannot resolve.

s (error)
: Segment error. The program attempted to initialize something in a segment that contains only uninitialized data.

u (error)
: A symbol is used but never defined. The symbol's name is displayed.

cpp Error Messages

string argument mismatch (error)
: The argument *string* does not match the type declared in the function's prototype. Either the function prototype or the argument should be changed.

#assert failure (error)
: The condition being tested in a **#assert** statement has failed.

at beginning of macro (error)
Macro replacement lists may contain tokens that are separated by **##**, but **##** cannot appear at the beginning or the end of the list. The tokens on either side of the **##** are pasted together into one token.

at end of macro (error)
Macro replacement lists may contain tokens that are separated by **##**, but **##** cannot appear at the beginning or the end of the list. The tokens on either side of the **##** are pasted together into one token.

string: cannot create (fatal)
The preprocessor **cpp** cannot create the output file *string* that it was asked to create. This often is due to a problem with the output device; check and make sure that it is not full and that it is working correctly.

string: cannot open (**cc0**, fatal)
The compiler cannot open the file *string* of source code that it was asked to read. **cpp** may not have been told the correct directory in which this file is to be found; check that the file is located correctly, and that the **-I** options, if any, are correct.

cannot open include file *string* (**cc0**, fatal)
The program asked for file *string*, which was not found in the same directory as the source file, nor in the default **include** directory specified by the environmental variable **INCDIR**, nor in any of the directories named in **-I** options given to the **cc** command.

conditional stack overflow (fatal)
A series of **#if** expressions is nested so deeply that it overflowed the allotted stack space. You should simplify this code.

#define argument mismatch (warning)
The definition of an argument in a **#define** statement does not match its subsequent use. One or the other should be changed.

#elif used without #if or #ifdef (error)
An **#elif** control line must be preceded by an **#if**, **#ifdef**, or **#ifndef** control line.

#elif used after #else (error)
An **#elif** control line cannot be preceded by an **#else** control line.

#else used without #if or #ifdef (error)
An **#else** control line must be preceded by an **#if**, **#ifdef**, or **#ifndef** control line.

#endif used without #if or #ifdef (error)
An **#endif** control line must be preceded by an **#if**, **#ifdef**, or **#ifndef** control line.

EOF in comment (fatal)
Your source file appears to end in mid-comment. The file of source code may have been truncated, or you failed to close a comment; make sure that each open-comment symbol '/*' is balanced with a close-comment symbol '*/'.

EOF in macro *string* invocation (error)
Your source file appears to end in a macro call. The source file may be been truncated.

EOF in midline (warning)
Check to see that your source file has not been truncated accidentally.

EOF in string (**cpp** ,error)
Your file appears to end in the middle of a quoted string literal. Check to see that your source file has not been truncated accidentally. Also, check that you did not accidentally embed a **<ctrl-Z>** in the line.

#error: *string* (fatal)
An **#error** control line has been expanded, printing the remaining tokens on the line and terminating the program.

error in #define syntax (error)
The syntax of a **#define** statement is incorrect. See the Lexicon entry for **#define** for more information.

error in #include syntax (error)
An **#include** directive must be followed by a string enclosed by either quotation marks (" ") or angle brackets (<>). Anything else is illegal.

identifier *string* has too many arguments (error)
Too many actual parameters have been provided.

illegal control line (error)
A '#' is followed by a word that the compiler does not recognize.

illegal cpp character (*n* decimal) (error)
The character noted cannot be processed by **cpp**. It may be a control character or a non-ASCII character.

illegal use of defined (error)
The construction **defined**(*token*) or **defined** *token* is legal only in **#if**, **#elif**, or **#assert** expressions.

string in #if (error)
A syntax error occurred in a **#if** declaration. *string* describes the error in detail.

include stack overflow (fatal)
A set of **#include** statements is nested so deeply that the allotted stack space cannot hold them. Examines the files for a loop. You should try to fold some of the header files into one, instead of having them call each other.

macro body too long (fatal)
The size of the macro in question exceeds the limit designed into the preprocessor. Try to shorten or split the macro.

macro expansion buffer overflow in *string* (fatal)
A macro call has expanded into more characters than **cpp** can handle. Try to shorten the macro, or break it up.

macro *string* redefined (error)
The program redefined the macro *string*.

macro *string* requires arguments (error)
The macro calls for arguments that the program has not supplied.

macros nested *number* deep, loop likely (error)
Macros call each other *number* times; you may have inadvertently created an infinite loop. Try to simplify the program.

missing #endif (error)
An **#if**, **#ifdef**, or **#ifndef** statement was not closed with an **#endif** statement.

missing output file (fatal)
: The preprocessor **cpp** found a **-o** option that was not followed by a file name for the output file.

multiple #else's (error)
: An **#if**, **#ifdef**, or **#ifndef** expression can be followed by no more than one **#else** expression.

nested comment (warning)
: The comment introducer sequence '/*' has been detected within a comment. Comments do not nest.

new line in *string* literal (error)
: A newline character appears in the middle of a string. If you wish to embed a newline within a string, use the character constant '\n'. If you wish to continue the string on a new line, insert a backslash '\' before the new line.

newline in macro argument (warning)
: A macro argument contains a newline character. This may create trouble when the program is run.

out of space (**cc0**, **cc1**, **cc2**, **cc3**, fatal)
: The compiler ran out of space while attempting to compile the program. To remove this error, examine your source and break up any functions that are extraordinarily large.

parameter must follow # (error)
: Macro replacement lists may contain # followed by a macro parameter name. The macro argument is converted to a string literal.

preprocessor assertion failure (warning)
: A **#assert** directive that was tested by the preprocessor **cpp** was found to be false.

string redefined (error)
: **cpp** macros should not be redefined. You should check to see that you are not **#include**ing two different versions of a file somehow, or attempting to use the same macro name for two different purposes.

too many arguments in a macro (fatal)
: The program uses more than the allowed ten arguments with a macro.

too many directories in include list (fatal)
: The program uses more than the allowed ten **#include** directories.

string: unknown option (fatal)
: The preprocessor **cpp** does not recognize the option *string*. Try re-typing the **cc** command line.

cc0 Error Messages

ambiguous reference to "*string*" (error)
: *string* is defined as a member of more than one **struct** or **union**, is referenced via a pointer to one of those **structs** or **unions**, and there is more than one offset that could be assigned.

argument list has incorrect syntax (error)
: The argument list of a function declaration contains something other than a comma-separated list of formal parameters.

array bound must be a constant (error)
An array's size can be declared only with a constant; you cannot declare an array's size by using a variable. For example, it is correct to say **foo[5]**, but illegal to say

```
bar = 5;
foo[bar];
```

array bound must be positive (error)
An array must be declared to have a positive number of elements. The array flagged here was declared to have a negative size, e.g., **foo[-5]**.

array bound too large (error)
The array is too large to be compiled with 16-bit index arithmetic. You should devise a way to divide the array into compilable portions.

array row has 0 length (error)
This message can be triggered by either of two problems. The first problem is declaring an array to have a length of zero; e.g., **foo[0]**. The second problem is failing to declare the size of a dimension *other than the first* in a multi-dimensional array. C allows you to declare an indefinite number of array elements of *n* bytes each, but you cannot declare *n* array elements of an indefinite length. For example, it is correct say **foo[][5]** but illegal to say **foo[5][]**.

bad argument storage class (error)
An argument was assigned a storage class that the compiler does not recognize. The only valid storage class is **register**.

bad external storage class (error)
An **extern** has been declared with an invalid storage class, e.g., **register** or **auto**.

bad field width (error)
A field width was declared either to be negative or to be larger than the object that holds it. For example, **char foo:9** or **char foo:-1** will trigger this error.

bad filler field width (error)
A filler field width was declared either to be negative or to be larger than the object that holds it. For example, **char foo:9** or **char foo:-1** will trigger this error.

bad flexible array declaration (error)
A flexible array is missing an array boundary; e.g., **foo[5][]**. C permits you to declare an indefinite number of array elements of *n* bytes each, but you cannot declare an array to have *n* elements of an indefinite number of bytes each.

break not in a loop (error)
A **break** occurs that is not inside a loop or a **switch** statement.

call of non function (error)
What the program attempted to call is not a function. Check to make sure that you have not accidentally declared a function as a variable; e.g., typing **char *foo;** when you meant **char *foo();**.

cannot add pointers (error)
The program attempted to add two pointers. **int**s or **long**s may be added to or subtracted from pointers, and two pointers to the same type may be subtracted, but no other arithmetic operations are legal on pointers.

cannot apply unary '&' to a register variable (error)
Because register variables are stored within registers, they do not have addresses, which means that the unary **&** operator cannot be used with them.

cannot apply unary '&' to an alien function (error)
The unary '&' operator cannot be used with any function that has been declared to be of type **alien**. **alien** functions cannot be called by pointers.

cannot cast double to pointer (error)
The program attempted to cast a **double** to a pointer. This is illegal.

cannot cast pointer to double (error)
The program attempted to cast a pointer to a **double**. This is illegal.

cannot cast structure or union (error)
The program attempted to cast a **struct** or a **union**. This is illegal.

cannot cast to structure or union (error)
The program attempted to cast a variable to a **union** or **struct**. This is illegal.

cannot declare array of functions (error)
For example, the declaration **extern int (*f)[]();** declares **f** to be an array of pointers to functions that return **int**s. Arrays of functions are illegal.

cannot declare flexible automatic array (error)
The program does not explicitly declare the number of elements in an automatic array.

cannot initialize fields (error)
The program attempted to initialize bit fields within a structure. This is not supported.

cannot initialize unions (error)
The program attempted to initialize a **union** within its declaration. **union**s cannot be initialized in this way.

case not in a switch (error)
The program uses a **case** label outside of a **switch** statement. See the Lexicon entry for **case**.

character constant overflows long (error)
The character constant is too large to fit into a **long**. It should be redefined.

character constant promoted to long (warning)
A character constant has been promoted to a **long**.

class not allowed in structure body (error)
A storage class such as **register** or **auto** was specified within a structure.

compound statement required (error)
A construction that requires a compound statement does not have one, e.g., a function definition, array initialization, or **switch** statement.

constant expression required (error)
The expression used with a **#if** statement cannot be evaluated to a numeric constant. It probably uses a variable in a statement rather than a constant.

constant "*number*" promoted to long (warning)
The compiler promoted a constant in your program to **long**; although this is not strictly illegal, it may create problems when you attempt to port your code to another system, especially if the constant appears in an argument list.

constant used in truth context (strict)
: A conditional expression for an **if**, **while**, or **for** statement has turned out to be always true or always false. For example, **while(1)** will trigger this message.

construction not in Kernighan and Ritchie (strict)
: This construction is not found in *The C Programming Language*; although it can be compiled by COHERENT, it may not be portable to another compiler.

continue not in a loop (error)
: The program uses a **continue** statement that is not inside a **for** for **while** loop.

declarator syntax (error)
: The program used incorrect syntax in a declaration.

default label not in a switch (error)
: The program used a **default** label outside a **switch** construct. See the Lexicon entry for **default**.

divide by zero (warning)
: The program will divide by zero if this code is executed. Although the program can be parsed, this statement may create trouble if executed.

duplicated case constant (error)
: A **case** value can appear only once in a **switch** statement. See the Lexicon entries for **case** and **switch**.

empty switch (warning)
: A **switch** statement has no **case** labels and no **default** labels. See the Lexicon entry for **switch**.

error in enumeration list syntax (error)
: The syntax of an enumeration declaration contains an error.

error in expression syntax (error)
: The parser expected to see a valid expression, but did not find one.

exponent overflow in floating point constant (warning)
: The exponent in a floating point constant has overflowed. The compiler has set the constant to the maximum allowable value, with the expected sign.

exponent underflow in floating point constant (warning)
: The exponent in a floating point constant has underflowed. The compiler has set the constant to zero, with the expected sign.

external syntax (error)
: This could be one of several errors, most often a missing '{'.

file ends within a comment (error)
: The source file ended in the middle of a comment. If the program uses nested comments, it may have mismatched numbers of begin-comment and end-comment markers. If not, the program began a comment and did not end it, perhaps inadvertently when dividing by ******something*, e.g., **a=b/*cd;**.

function cannot return a function (error)
: The function is declared to return another function, which is illegal. A function, however, can return a *pointer* to a function, e.g., **int (*signal(n, a))()**.

function cannot return an array (error)
: A function is declared to return an array, which is illegal. A function, however, can return a pointer to a structure or array.

functions cannot be parameters (error)
The program uses a function as a parameter, e.g., **int q(); x(q);**. This is illegal.

identifier "*string*" is being redeclared (error)
The program declares variable *string* to be of two different types. This often is due to an implicit declaration, which occurs when a function is used before it is explicitly declared. Check for name conflicts.

identifier "*string*" is not a label (error)
The program attempts to **goto** a nonexistent label.

identifier "*string*" is not a parameter (error)
The variable "*string*" did not appear in the parameter list.

identifier "*string*" is not defined (error)
The program uses identifier *string* but does not define it.

identifier "*string*" not usable (error)
string is probably a member of a structure or **union** which appears by itself in an expression.

illegal character constant (error)
A legal character constant consists of a a backslash '\' followed by **a**, **b**, **f**, **n**, **r**, **t**, **v**, **x**, or up to three octal digits.

illegal character (*number* decimal) (error)
A control character was embedded within the source code. *number* is the decimal value of the character.

illegal # construct (error)
The parser recognizes control lines of the form *#line_number* (decimal) or *#file_name*. Anything else is illegal.

illegal integer constant suffix (error)
Integer constants may be suffixed with **u**, **U**, **l**, or **L** to indicate **unsigned**, **long**, or **unsigned long**.

illegal label "*string*" (error)
The program uses the keyword *string* as a **goto** label. Remember that each label must end with a colon.

illegal operation on "void" type (error)
The program tried to manipulate a value returned by a function that had been declared to be of type **void**.

illegal structure assignment (error)
The structures have different sizes.

illegal subtraction of pointers (error)
A pointer can be subtracted from another pointer only if both point to objects of the same size.

illegal use of a pointer (error)
A pointer was used illegally, e.g., multiplied, divided, or &-ed. You may get the result you want if you cast the pointer to a **long**.

illegal use of a structure or union (error)
You may take the address of a **struct**, access one of its members, assign it to another structure, pass it as an argument, and return. All else is illegal.

ERROR MESSAGES

illegal use of floating point (error)
A **float** was used illegally, e.g., in a bit-field structure.

illegal use of "void" type (error)
The program used **void** improperly. Strictly, there are only **void** functions; COHERENT also supports the cast to **void** of a function call.

illegal use of void type in cast (error)
The program uses a pointer where it should be using a variable.

inappropriate signed (error)
The **signed** modifier may only be applied to **char**, **short**, **int**, or **long** types.

inappropriate "long" (error)
Your program used the type **long** inappropriately.

inappropriate "short" (error)
Your program used the type **short** inappropriately.

inappropriate "unsigned" (error)
Your program used the type **unsigned** inappropriately.

indirection through non pointer (error)
The program attempted to use a scalar (e.g., a **long** or **int**) as a pointer. This may be due to not de-referencing the scalar.

initializer too complex (error)
An initializer was too complex to be calculated at compile time. You should simplify the initializer to correct this problem.

integer pointer comparison (strict)
The program compares an integer or **long** with a pointer without casting one to the type of the other. Although this is legal, the comparison may not work on machines with non-integer size pointers, e.g., Z8001 or LARGE-model on the i8086 family, or on machines with pointers larger than **int**s, e.g., the M68000 family of microprocessors.

integer pointer pun (strict)
The program assigns a pointer to an integer, or vice versa, without casting the right-hand side of the assignment to the type of the left-hand side. For example,

```
char *foo;
long bar;
foo = bar;
```

Although this is permitted, it is often an error if the integer has less precision than the pointer does. Make sure that you properly declare all functions that returns pointers.

internal compiler error (fatal)
The program produced a state that should not happen during compilation. Try to localize the offending statement if at all possible. Forward a minimal program that exhibits the error, preferably on a machine-readable medium, to Mark Williams Company, together with the version number of the compiler, the command line used to compile the program, and the system configuration. For immediate advice during business hours, telephone Mark Williams Company technical support.

"*string*" is a enum tag (error)

"*string*" is a struct tag (error)
"*string*" is a union tag (error)
string has been previously declared as a tag name for a **struct**, **union**, or **enum**, and is now being declared as another tag. Perhaps the structure declarations have been included twice.

"*string*" is not a tag (error)
A **struct** or **union** with tag *string* is referenced before any such **struct** or **union** is declared. Check your declarations against the reference.

"*string*" is not a typedef name (error)
string was found in a declaration in the position in which the base type of the declaration should have appeared. *string* is not one of the predefined types or a **typedef** name. See the Lexicon entry on **typedef** for more information.

"*string*" is not an "enum" tag (error)
An **enum** with tag *string* is referenced before any such **enum** has been declared. See the Lexicon entry for **enum** for more information.

class "*string*" [*number*] is not used (strict)
Your program declares variable *string* or *number* but does not use it.

label "*string*" undefined (error)
The program does not declare the label *string*, but it is referenced in a **goto** statement.

left side of "*string*" not usable (error)
The left side of the expression *string* should be a pointer, but is not.

lvalue required (error)
The left-hand value of a declaration is missing or incorrect. See the Lexicon entries for **lvalue** and **rvalue**.

member "*string*" is not addressable (error)
The array *string* has exceeded the machine's addressing capability. Structure members are addressed with 16-bit signed offsets on most machines.

member "*string*" is not defined (error)
The program references a structure member that has not been declared.

mismatched conditional (error)
In a '?:' expression, the colon and all three expressions must be present.

missing "(" (error)
The **if**, **while**, **for**, and **switch** keywords must be followed by parenthesized expressions.

missing "=" (warning)
An equal sign is missing from the initialization of a variable declaration. Note that this is a warning, not an error: this allows COHERENT to compile programs with "old style" initializers, such as **int i 1**. Use of this feature is strongly discouraged, and it will disappear when the ANSI standard for the C language is adopted in full.

missing "," (error)
A comma is missing from an enumeration member list.

missing ":" (error)
A colon ':' is missing after a **case** label, after a default label, or after the '?' in a '?'-':' construction.

missing ";" (error)
A semicolon ';' does not appear after an external data definition or declaration, after a **struct** or **union** member declaration, after an automatic data declaration or definition, after a statement, or in a **for(;;)** statement.

missing "]" (error)
A right bracket ']' is missing from an array declaration, or from an array reference; for example, **foo[5**.

missing "{" (error)
A left brace '{' is missing after a **struct** *tag*, **union** *tag*, or **enum** *tag* in a definition.

missing "}" (error)
A right brace '}' is missing from a **struct**, **union**, or **enum** definition, from an initialization, or from a compound statement.

missing "while" (error)
A **while** command does not appear after a **do** in a **do-while()** statement.

missing label name in goto (error)
A **goto** statement does not have a label.

missing member (error)
A '.' or '->' is not followed by a member name.

missing right brace (error)
A right brace is missing at end of file. The missing brace probably precedes lines with errors reported earlier.

missing "*string*" (error)
The parser **cc0** expects to see token *string*, but sees something else.

missing semicolon (error)
External declarations should continue with ',' or end with ';'.

missing type in structure body (error)
A structure member declaration has no type.

multiple classes (error)
An element has been asigned to more than one storage class, e.g., **extern register**.

multiple types (error)
An element has been assigned more than one data type, e.g., **int float**.

nonterminated string or character constant (error)
A line that contains single or double quotation marks left off the closing quotation mark. A newline in a string constant may be escaped with '\'.

number has too many digits (error)
A number is too big to fit into its type.

only one default label allowed (error)
The program uses more than one **default** label in a **switch** expression. See the Lexicon entries for **default** and **switch** for more information.

out of tree space (fatal)
The compiler allows a program to use up to 350 tree nodes; the program exceeded that allowance.

parameter *string* is not addressable (error)
The parameter has a stack frame offset greater than 32,767. Perhaps you should pass a pointer instead of a structure.

potentially nonportable structure access (strict)
A program that uses this construction may not be portable to another compiler.

return type/function type mismatch (error)
What the function was declared to return and what it actually returns do not match, and cannot be made to match.

return(e) illegal in void function (error)
A function that was declared to be type **void** has nevertheless attempted to return a value. Either the declaration or the function should be altered.

risky type in truth context (strict)
The program uses a variable declared to be a pointer, **long**, **unsigned long**, **float**, or **double** as the condition expression in an **if**, **while**, **do**, or '?'-':'. This could be misinterpreted by some C compilers.

size of *string* overflows size_t (strict)
A string was so large that it overran an internal compiler limit. You should try to break the string in question into several small strings.

size of union "*string*" is not known (error)
A pointer to a **struct** or **union** is being incremented, decremented, or subjected to array arithmetic, but the **struct** or **union** has not been defined.

size of *string* too large (error)
The program declared an array or **struct** that is too big to be addressable, e.g., **long a[20000];** on a machine that has a 64-kilobyte limit on data size and four-byte **longs**.

sizeof truncated to unsigned (warning)
An object's **sizeof** value has lost precision when truncated to a **size_t** integer.

sizeof(*string*) set to *number* (warning)
The program attempts to set the value of *string* by applying **sizeof** to a function or an **extern**; the compiler in this instance has set *string* to *number*.

storage class not allowed in cast (error)
The program **casts** an item as a **register**, **static**, or other storage class.

string initializer not terminated by NUL (warning)
An array of **chars** that was initialized by a string is too small in dimension to hold the terminating NUL character. For example, **char foo[3] = "ABC"**.

structure "*string*" does not contain member "*m*" (error)
The program attempted to address the variable *string.m*, which is not defined as part of the structure *string*.

structure or union used in truth context (error)
The program uses a structure in an **if**, **while**, or **for**, or '?:' statement.

switch of non integer (error)
The expression in a **switch** statement is not type **int** or **char**. You should cast the **switch** expression to an **int** if the loss of precision is not critical.

too many adjectives (error)
: A variable's type was described with too many of **long**, **short**, or **unsigned**.

too many arguments (fatal)
: No function may have more than 30 arguments.

too many initializers (error)
: The program has more initializers than the space allocated can hold.

too many structure initializers (error)
: The program contains a structure initialization that has more values than members.

trailing "," in initialization list (warning)
: An initialization statement ends with a comma, which is legal.

type clash (error)
: The parser expected to find matching types but did not. For example, the types of **e1** and **e2** in **(x) ? e1 : e2** must either both be pointers or neither be pointers.

type of function "*string*" adjusted to *string* (warning)
: This warning is given when the type of a numeric constant is widened to **unsigned**, **long**, or **unsigned long** to preserve the constant's value. The type of the constant may be explicitly specified with the **u** or **L** constant suffixes.

type of parameter "*string*" adjusted to *string* (warning)
: The program uses a parameter that the C language says must be adjusted to a wider type, e.g., **char** to **int** or **float** to **double**.

type required in cast (error)
: The type is missing from a cast declaration.

unexpected end of enumeration list (error)
: An end-of-file flag or a right brace occurred in the middle of the list of enumerators.

unexpected EOF (**cc1**, **cc2**, **cc3**, fatal)
: **EOF** occurred in the middle of a statement. The temporary file may have been corrupted or truncated accidentally. Check your disk drive to see that it is working correctly.

union "*string*" does not contain member *m* (error)
: The program attempted to address the variable string *m*, which is not defined as part of the structure *string*.

zero modulus (warning)
: The program will perform a modulo operation by zero if the code just parsed is executed. Although the program can be parsed, this statement may create trouble if executed.

cc1 Error Messages

associative expression too complex (fatal)
: An expression that uses associative binary operators (e.g., '+') has too many operators; for example, **i=i1+i2+i3+ . . . +i30;**. You should simplify the expression.

expression too complex (fatal)
: The code generator cannot generate code for an expression. You should simplify your code.

internal compiler error (fatal)
The program produced a state that should not happen during compilation. Try to localize the offending statement if at all possible. Forward a minimal program that exhibits the error, preferably on a machine-readable medium, to Mark Williams Company, together with the version number of the compiler, the command line used to compile the program, and the system configuration. For immediate advice during business hours, telephone Mark Williams Company technical support.

misplaced ":" operator (error)
The program used a colon without a preceding question mark. It may be a misplaced label.

switch overflow (fatal)
The program has more than ten nested **switches**.

too many cases (fatal)
The program cannot allocate space to build a **switch** statement.

cc2 Error Messages

string: cannot reopen (fatal)
The optimizer in **cc2** cannot reopen a file with which it has worked. Make sure that your mass storage device is working correctly and that it is not full.

internal compiler error (fatal)
The program produced a state that should not happen during compilation. Try to localize the offending statement if at all possible. Forward a minimal program that exhibits the error, preferably on a machine-readable medium, to Mark Williams Company, together with the version number of the compiler, the command line used to compile the program, and the system configuration. For immediate advice during business hours, telephone Mark Williams Company technical support.

write error on output object file (fatal)
cc2 could not write the relocatable object module. Most likely, your mass storage device has run out of room. Check to see that your disk drive or hard disk has enough room to hold the object module, and that it is working correctly.

Linker Error Messages

address wraparound (fatal)
A segment of the program has exceeded the size allowed by the microprocessor's architecture.

baddisk: disk error (fatal)
ld either cannot read or cannot write to the mass-storage device. Check the disk you are using to see that it is working correctly.

cannot create *string* (fatal)
The linker **ld** cannot create the output file it was requested to create. This often is due to a problem with the output device; check and make sure that it is working correctly and is not full.

cannot open *string* (seg *number*) (fatal)
The linker **ld** cannot open the object module that it was asked to read. Make sure that the storage device is working correctly, and that **ld** has been given the correct names of the file and of the directory in which it is stored.

can't open lib*string*.a (fatal)
: The linker **ld** cannot open a library that it has been asked to link into your program. Make sure that you named the library correctly and that the **LIBPATH** is set correctly if you used the **-l** option to the **cc** command line.

can't open temp file (fatal)
: The linker **ld** cannot open a temporary file. Make sure that your mass storage device is working correctly.

can't read *string* (fatal)
: The linker **ld** cannot read the file named. Make sure that your mass storage device is working correctly, and that **ld** has been given the correct names of the file and of the directory in which it is stored.

disk error (fatal)
: The linker **ld** encountered a problem with the storage device when it attempted to read or write a file. Check that the disk is working correctly.

no input found (fatal)
: The **ld** command line names no object or archive files to link.

out of space (fatal)
: **malloc** could not allocate adequate space in memory for the linker **ld** to work.

outdated ranlib (warning)
: The date stamp on the library file is younger than that in the ranlib header. If the library has been altered, the ranlib can be updated with the archiver **ar**; see the Lexicon entry on **ar** to see how this is done. If the library has not been altered, this message may be due to an installation error; see the Lexicon entry on **ranlib** for more information.

fsck Error Messages

The COHERENT command **fsck** checks the COHERENT file systems. This command produces an especially rich set of error messages, both to keep you abreast of its actions and to warn you of potential problems with a file or file system.

fsck can correct most of the common error conditions it detects; however, before it will ask for your approval before it makes any changes that modify a file system. Therefore, if it detects an error that it can correct, it will stop and ask your permission.

The following describes **fsck**'s error messages and questions. The error messages fall into two categories: *warnings*, which describe something possibly wrong with a file; and **fatals**, which indicate that something has gone wrong with a file system or with **fsck** with which **fsck** cannot cope. Each question describes the condition in question; here, it is followed by advice on what is probably the correct response.

Initialization

Can't open checklist file: /etc/checklist (Fatal)

Too many file systems in checklist file: /etc/checklist (Fatal)

file is not a block or character device; OK? [yes/no]: (Question)
: You are attempting to **fsck** a file that is not a block or character device. If you are certain it is a file system, then answer yes to continue.

Can't open: *file system (Warning)*

Can't stat: *file system (Warning)*

Size check: fsize *blocks* isize *first non-i-node block* (Warning)

Too large free block count (Warning)

Too large free i-node count (Warning)

fsck: *file system*: Bad Super Block: *number* (Warning)

file system mounted on *point* as of *time* (Message)

file system unmounted. Last mounted on *point*. (Message)

Phase 1: Check Blocks and Sizes

Unknown File Type i-number = *number* (Clear) [yes/no]: (Question)
The mode field in the specified i-node is unknown. If you wish, you can clear the named i-node.

Excessive Bad Blocks i-number = *number* (Continue) [yes/no] (Question)
The specified i-node references an excessive number of bad blocks. You can continue with the **fsck** (at the next i-node), or abort.

DUP Table Overflow (Continue) [yes/no] (Question)
The table of duplicately referenced disk blocks has overflowed. You can continue with the **fsck** (as best as it is able), or abort.

Excessive Dup Blocks i-number = *number* (Continue) [yes/no] (Question)
The specified i-node references an excessive number of duplicate blocks. You can continue with the **fsck** (at the next i-node), or abort.

Bad block *number*, i-number = *number* (Warning)

Dup Block *number*, i-number = *number* (Warning)

Directory Misaligned i-number = *number* (Warning)

Possible Directory Size Error i-number = *number* (Warning)

Possible File Size Error i-number = *number* (Warning)

Phase 1b: Rescan for more Duplicates

Dup Block *number*, i-number = *number* (Warning)

Phase 2: Check Path Names

Root i-node is unallocated. Terminating (FATAL)

File System Read-Only (NO WRITE) (FATAL)

Can't malloc memory, phase 2 (FATAL)

Fixblock error. (FATAL)

Tried to checkpath i-node *number* which is not dir. (FATAL)

Root i-node is not a directory (FIX) [yes/no] (Question)
The root i-node must be a directory. **fsck** is asking whether you wish to fix this. If not, then **fsck** will abort.

Dup/Bad blocks in root i-node (Continue) [yes/no] (Question)
The root i-node has bad or duplicate blocks. This may require a guru to fix properly. **fsck** is asking whether you want it to continue. If not, then **fsck** will abort.

I-number is out of range I=*file name* (Remove) [yes/no] (Question)
file has an i-node number that is out of range. **fsck** is asking if you wish to remove the stated file (which, after all, does not exist).

Unallocated *file* (Remove) [yes/no] (Question)
file's i-node is unallocated. **fsck** is asking if you wish to remove the stated file (which, after all, does not exist).

Bad or Dup blocks in *directory/file* (Remove) [yes/no] (Question)
The given file's i-node references bad or duplicately referenced blocks. **fsck** is asking if you wish to remove *file* from the directory.

Null name entry in block *number* in directory *name/i-node* (Warning)

Non null padded entry in block *number* in directory *name/i-node* (Warning)

Embedded slashes in entry in block *number* in directory *name/i-node* (Warning)

Inconsistent . entry in block *number* in directory *name/i-node* (Warning)

Inconsistent .. entry in block *number* in directory *name/i-node* (Warning)

Bad entry in block *number* in directory *name/i-node* (Warning)

i-number = *number* is in a bad inode block. (Warning)

I-node *number* is a multiply referenced directory i-node. (Warning)

Name too long. (Warning)

Phase 3: Check Connectivity

Unref Dir *name* (Reconnect) [yes/no] (Question)
The given directory's i-node is unreferenced. You are asked if you would like to reconnect the stated directory. If you answer yes, then the directory will be reconnected in directory **/lost+found** in the given file system. If not, it will remain unreferenced and you will be asked later if you would like to remove it.

Dir i-number = *number* connected. Parent was i-number = *number* (Warning)

Dir i-number = *number* connected. It has bad/dup blocks. (Warning)

Dir i-number = *number* connected. It has no .. entry. (Warning)

Sorry. No lost+found directory. (Warning)

Sorry. No space in lost+found directory. (Warning)

Phase 4: Check Reference Counts

Unref *i-node type file name* (Reconnect) [yes/no] (Question)
The given i-node is unreferenced. **fsck** is asking if you wish to reconnect it to the stated file. If you answer yes, then the file will be reconnected in directory **/lost+found** in the given file system. If not, it will remain unreferenced and you will be asked later if you would like to remove it.

Unref *i-node type file name* (Clear i-node) [yes/no] (Question)
The given i-node is unreferenced. **fsck** asks if you wish to clear the i-node completely. If you answer yes, the file is lost forever. You have already decided not to reconnect it, so there seems to be no reason to keep it anyway.

Bad/Dup blocks in *i-node type file name* (Clear i-node) [yes/no] (Question)
The given i-node contains bad or duplicately referenced blocks. You are asked if you would like to clear the inode completely. If you answer yes, then the file will be lost forever.

Link count discrepancy in *i-node type file name*

Count = *count*, should be *count* (Adjust) [yes/no] (Question)
The given i-node claims to have a different number of links than was actually found in the file system. You are asked if you wish to adjust the count found in the i-node. If you answer yes, then **fsck** will correct the i-node count.

Free i-node count wrong in superblock. (FIX) [yes/no] (Question)
The free i-node count in the superblock is incorrect. You should allow **fsck** to repair it unless you are a guru and have reason to believe that **fsck** should not use the redundancy in the file system (via all previously reported messages) to repair this crucial piece of data in the superblock.

Phase 5: Check Free List

Free Block count wrong in superblock. (FIX) [yes/no] (Question)
The free block count in the superblock is incorrect. You should allow **fsck** to repair it unless you are a guru and have reason to believe that **fsck** should not use the redundancy in the file system (via all previously reported messages) to repair this crucial piece of data in the superblock.

Excessive *bad/dup* blocks in free list (Continue) [yes/no] (Question)
This indicates that there are excessive bad or duplicately referenced blocks in the free list off of the superblock. This is a very bad condition. You should choose to continue, which will fall to phase 6 to salvage the free list. If you answer no, then **fsck** will abort.

Bad Free List (SALVAGE) [yes/no] (Question)
fsck is asking if you want it to salvage the free list automatically. This is almost certainly a good thing to do.

Number Bad blocks in Free List (Warning)

Number Dup blocks in Free List (Warning)

Number Blocks missing (Warning)

ERROR MESSAGES

Phase 6: Salvage Free List

Invalid interleave factors in superblock. Default free-block list spacing assumed. (Warning)

Can't malloc space for interleave table. Free-block list is not rebuilt. (Warning)

Cleanup

Number files *number* blocks *number* free (Message)

Expect roughly *number* missing blocks next time fsck is run as a result of i-nodes being cleared. (Message)

***** File System *system* was modified ***** (Message)

***** BOOT Coherent (NO SYNC!) ***** (Message)
Do as the message says: reboot COHERENT *without* running the command **sync**.

General Messages

Bad action in virtual system (FATAL)

Can not Seek: Blk num: *number* (CONTINUE) [yes/no] (Question)
The given action could not be performed. If you choose to not continue, **fsck** will abort. If you choose to continue, the results may be unpredictable.

Can not Read: Blk num: *number* (CONTINUE) [yes/no] (Question)
The given action could not be performed. If you choose to not continue, **fsck** will abort. If you choose to continue, the results may be unpredictable.

Can not Write: Blk num: *number* (CONTINUE) [yes/no] (Question)
The given action could not be performed. If you choose to not continue, **fsck** will abort. If you choose to continue, the results may be unpredictable.

Cannot create temp file *name* (FATAL)

Cannot close Ram Disk Close /dev/rram1close (FATAL)

Cannot open Ram Disk Close /dev/rram1close (FATAL)

Cannot open read/write Ram Disk /dev/rram1 (FATAL)

Can't access ram disk /dev/rram1, use the -t option (FATAL)

Can't stat temp file *name* (FATAL)

Error seeking tmp file (FATAL)

Error writing tmp file (FATAL)

Error writing to tmp file (FATAL)

internal linktable corruption. (FATAL)

Invalid Response (FATAL)

Out of Range Block number: *number* (CONTINUE) [yes/no] (Question)
The given action could not be performed. If you choose to not continue, **fsck** will abort. If you choose to continue, the results may be unpredictable.

Possible file system on ram disk /dev/rram1, use the -t option (FATAL)

Ram disk close /dev/rram1close not mknoded properly (FATAL)

Ram disk /dev/rram1 not mknoded properly (FATAL)

Temp File must not be on file system to fsck (FATAL)

Too many links in i-node *number* (FATAL)

make Error Messages

The following gives the error messages that can be produced by **make**. Its message describe fatal conditions, errors, or warnings, as described above.

; after target or macroname (error)
: A semicolon appeared after a target name or a macro name.

Bad macro name (error)
: A bad macro name was used; for example, a macro name included a control character.

= in or after dependency (error)
: An equal sign '=' appeared within or followed the definition of a macro name or target file; for example, **OBJ=atod.o=factor.o** will produce this error.

Incomplete line at end of file (error)
: An incomplete line appeared at the end of the **makefile**.

Macro definition too long (error)
: The macro definition exceeds the limited designed into the preprocessor.

Multiple actions for *name* (error)
: A target is defined with more than one single-colon target line.

Multiple detailed actions for *name* (error)
: A target is defined with more than one single-colon target line.

Must use '::' for *name* (error)
: A double-colon target line was followed by a single-colon target line.

Newline after target or macroname (error)
: A newline character appears after a target name or a macro name.

'::' not allowed for *name* (error)
: A double-colon target line was used illegally; for example, after single-colon target line.

::: or : in or after dependency list (error)
: A triple colon is meaningless to and therefore illegal wherever it appears. A single colon may be used only in a target line (which is also called the *dependency list*), and nowhere else.

Out of core (adddep) (error)
: This results from a system problem. Try reducing the size of your **makefile**.

Out of range number input. (**resource**, warning)
: You attempted to use a numeric value that is out of range.

: To clear this message, press the left mouse button or any key.

ERROR MESSAGES

Out of space (error)
System problem. Try reducing the size of your **makefile**.

Out of space (lookup) (error)
System problem. Try reducing the size of your **makefile**.

Syntax error (error)
The syntax of a line is faulty.

Too many macro definitions (error)
The number of macros you have created exceeds the capacity of your computer to process them.

= without macro name or in token list (error)
An equal sign '=' can be used only to define a macro, using the following syntax: "MACRO=*definition*". An incomplete macro definition, or the appearance of an equal sign outside the context of a macro definition, will trigger this error message.

: without preceding target (error)
A colon appeared without a target file name, e.g., :*string*.

nroff Error Messages

The following gives **nroff**'s error messages, and hints about how to correct the situation. Errors are of two types: *simple errors*, which simply cause an error message to be printed on your screen; and *panics*, which causes processing to abort. Note that a panic will leave behind a half-written temporary file; you may wish to look at the end of it to see just how far processing proceeded, but otherwise it should be thrown away.

-f option requires file argument (fatal)

.bd not implemented yet

.co: unexpected EOF before *string* (error)

.dt not implemented yet

.el without .ie (error)

.fc not implemented yet

.hc not implemented yet

.hw not implemented yet

.hy not implemented yet

.ie nested more than *N* levels (error)
The **.ie**/**.el** combination can be nested only 15 levels deep.

.ie without matching .el (error)
Every **.ie** must be followed by an **.el**.

.lf: *string*, file "*string*" (error)
nroff could not load a font-width table from file *string*.

.lf: "*string*" is not a PCL font width table (error)
nroff expects a PCL font-width table, but file *string* is not in the PCL font-width format.

.lf: "*string*" is not a PostScript font width table (error)
: **nroff** expects a PostScript font-width table, but file *string* is not in the PostScript font-width format.

.lf: cannot load more than *N* fonts (error)
: **nroff** has a static limit on the number of font-width tables that can be loaded at one time.

.lf: cannot open file "*string*" (error)

.lf: requires fontname and filename (error)

.nm not implemented yet

.nn not implemented yet

.pi not implemented yet

.rb: cannot open file *string* (error)

.rb: no file specified (error)

.rf: requires name and new name (error)

\} without matching \{ (error)
: Every \} must be preceded by a \{.

arguments too long (error)

attempted zero divide (error)

attempted zero modulus (error)

bad adjustment type (error)

bad argument reference (error)

bad directive *N* (fatal)

bad font *N* (fatal)

bad font *N* at dev_font, nfonts=*N* (fatal)

bad font *N*, nfonts=*N* (fatal)

bad pattern (fatal)

bad tab stop (error)

bad tab stop (error)

botch: fontname(*N*) (fatal)
: **nroff** cannot handle font *N* and must abort processing.

botch: swdmul=*N* psz=*N* swddiv=*N* (fatal)
: An undefined error has occurred within **nroff**. The printed numbers give the value of **nroff**'s internal registers. If such an error occurs regularly when you process a given piece of text, please send the text in question and a copy of the error message to Mark Williams technical support.

bracket building not implemented yet

cannot create temp file (fatal)

ERROR MESSAGES

cannot dehyphenate (fatal)

cannot end diversion (error)

cannot find current file (error)

cannot find font *XX* (error)

cannot find font *N* (error)

cannot find register *string* (error)

cannot open *string* (error)

cannot open file "*string*" (error)

cannot pop environment (error)

cannot read environment (fatal)

cannot remove *string* (error)

cannot reopen temp file (fatal)

cannot write environment (fatal)

delimiter argument too large (error)

diversion buffer odd alignment (fatal)

environment does not exist (error)

environments stacked too deeply (error)

field with too large (error)

file "*string*" not found (error)

flushd -- current diversion null (fatal)

font position out of range (error)

fonts.r not found (fatal)

illegal hex digit (error)

The escape sequence **X***NN* prints a character by its literal hexadecimal value. This should be used when processing characters that are not normally printable on the terminal screen. Digit *N* can be the numerals '0' through '9', the letters 'a' through 'f', or the letters 'A' through 'F'. All other characters will trigger this error.

illegal option: *string* (fatal)

incomplete macro in trap (fatal)

line buffer overflow (fatal)

no room for new font name *XX* (error)

out of space - memory *string* (fatal)

request '*string*' not found (error)

section *N* of title too large (error)

special character *XX* not found (error)

syntax error (error)

temporary file write error (fatal)

too many tab stops (error)

unexpected end of file (fatal)

unknown macro/register type *N* (fatal)

vertical line drawing not implemented yet

word buffer overflow (fatal)

Index

to _

. 379
. 380
#define 105, 381
#elif 382
#else 383
#endif 383
#if 383
#ifdef 383
#ifndef 384
#include 104, 384
#line 385
#undef 385

$ 323, 327

& 42, 318
&& 330

) 334

* 320-321, 334
*) 334

, (comma) 41

- (hyphen) 41

/ 322
/ (slash) 27
/bin 12, 27, 327
/coherent 444
/dev 27
/dev/console 663
/dev/tty/?/? 663
/drv 27
/etc 28
/etc/drvld 27
/etc/group 479
/etc/passwd 28, 47-48
/etc/rc 393, 445, 663
/etc/ttys 28, 42, 663
/etc/update 26, 445
/etc/utmp 445, 663
/etc/wtmp 445
/lib 28
/u 28
/usr 28
/usr/adm 28
/usr/adm/acct 56, 393
/usr/adm/savacct 56
/usr/adm/wtmp 53, 389
/usr/bin 28
/usr/games 28
/usr/games/lib/fortunes 28
/usr/include 28
/usr/lib/crontab 28, 41
/usr/lib/lib.b 439
/usr/man 28
/usr/msgs 28
/usr/pub 28
/usr/spool 28
/usr/wtmp 663

24-hour time 41

: (colon) 327

; 317
;; (double semicolon) 334

< 30
<ctrl-D> 9, 35
<ctrl-H> 5
<erase> 5, 40
<interrupt> 5
<kill> 5, 40
<Return> 5, 37

= 409

> 14, 29, 327, 330

? 321

[. 322, 408

.sig.mail 738

] 322, 408

__DATE__ 385
__FILE__ 386
__LINE__ 386
__STDC__ 386
__TIME__ 387
_exit() 387

A

abort() 388
abs() 388
ac 389
access permission 20
access() 390
access.h 391
accounting 52
  login 53
  process 54
  reports 52
  starting login 53
  starting process 56
acct() 391
acct.h 392

accton 393
acos() 393
action.h 394
address 98, 103, 394
aggregate 403
aha154x 395
alarm() 397
alias 397
aliases 398
alignment 400
alloc.h 400
altering stack size 97
apostrophe 321
ar 400
ar.h 401
archive
  extracting from compressed 1042
archive file, format 401
arena 402, 1043
argc 104, 402
argument 100
arguments 6
arguments, variable number of 1028
argv 104, 403
array 403
array, search 447
as 97, 404
  error messages 1045
ASCII 423
ASCII file 602
ascii.h 425
asctime() 425
asin() 426
ASKCC 427
assembler
  80286 instructions 416
  8086 instructions 411
  8087 op codes 417
  address descriptors 410
  blank between tokens 406
  blank character 406
  C compatibility 419
  comments 406
  current location counter 407
  diagnostics 422
  directives 409
  error messages 1045
  expressions 407
  function arguments 419
  function return values 420
  function-calling conventions 419
  identifiers 405
  labels name 409
  labels temporary 409
  names 405
  naming conventions 419
  operators 408
  options -g 405
  options -l 405
  options -o 405
  private data 407
  private instruction 407
  register names 405
  registers 420
  sections 406
  shared data 407
  shared instruction 407
  statements 409
  strings section 407
  symbol table 407
  symbols 405
  tabulation between tokens 406
  tokens 405
  type propagation 408
  uninitialized data 407
  uninitialized instruction 407
assembler directive
  ascii 410
  blkb 410
  blkw 410
  bssd 409
  bssi 409
  byte 410
  even 410
  globl 410
  odd 410
  page 410
  prvd 409
  prvi 409
  shrd 409
  shri 409
  strn 409
  symt 409
  title 410
  word 410
assembler expressions
  grouping 408
assembler operators
  * 409
  + ... 408
  binary - 409
  unary - 408
  ~ 408
assembler statements
  assignment 409
  null 409
assembler types
  absolute 408
  register 408
  relocatable 408
  undefined 408
assembly language 44, 98
assembly-language generator 94
assembly-language programs 97
assert() 427
assert.h 427
assertion, check at run time 427
at 428-429
atan() 430
atan2() 431

ATclock 431
atod.c 96
atof() 431
atoi() 432
atol() 432
atrun 433
auto 433
awk 434
  tutorial 57

B

background
  running programs in 784
background process 318
background, execution in 685, 904
backup 610
backup files 511, 552, 966, 1010
backups
  strategies 49
bad 436
badscan 436
banner 437
basename 437
Bathsheba 238
baud rate 491
  table 900
bc 437
  assignment 77
  exponentiation operator 75
  library 92
  tutorial 75
BCD format 609
bibliography 113
binary coded decimal 609
binary files 602
binary search 447
bind 440
bit 440
bit map 441
bit-fields 440
block 441
block, disk 19
block-special device 441
boot 442
  device 444
boot.fha 444
bootable floppy disk 442
booting 444
boottime 446
boottime, check file system 478
boottime, mount file system 781
boottime, standard chores 866
Bourne shell 904
brace 45, 325
braces 100
brc 446
break 446
break a string into tokens 949-950
brk() 447
bsearch() 447
bssd 407
bssi 407
buf.h 449
buffer 449
build 450
builtin 450
byte 450
byte ordering 450

C

C 44-45
c 452
C
  program linker 45
C keywords 452
C language 453
  tutorial 93
C preprocessor 94, 455
  error messages 1045
C programming
  introduction 97
cabs() 456
cal 457
calendar 457
calling conventions 458
calloc() 462
candaddr() 463
candev() 463
canino() 463
canint() 464
canlong() 464
canon.h 464
canshort() 466
cansize() 466
cantime() 466
canvaddr() 467
carriage return 100
case 333-334, 467
case sensitivity
  in file names 10
  in shell variable 324
cast 468
cat 6, 14, 29, 468
caveat utilitor 469
cc 44-45, 93, 469
  error messages 1044
  MicroEMACS mode 229
cc0 . . . . . 94, 473
  error messages 1048
cc1 . . . . . 94, 473, 1057
cc2 . . . . . 94, 473, 1058
cc3 . . . . . 94, 474
cd 15, 327, 474
ceil() 474
cgrep 475
char 103, 476
character, copy 756, 759
character, fill an area with 761

character, reverse search for 947
character, search for in region of memory 757
character, search for in string 939, 947
character, search string for 947
character-special device 441
chars.h 477
chase 477
chdir() 478
check 478
check assertion at run time 427
checklist 478
chgrp 479
chmod 21, 319, 480
chmod() 479
choices
  in case statements 334
chown 482
chown() 481
chroot() 482
ckermit 482
clear 488
clearerr() 488
close() 488
clri 488
cmp 328-329, 489
code generator 94
code, conditional inclusion, end 383
code, include code conditionally . . . 383-384
code, include conditionally 383
COHERENT
  description 1
  hardware requirements 1
  on same hard drive as MS-DOS 594
  rebooting 7
  shutting down 7
COHERENT file format 545
COHERENT system
  error messages 1043
COHERENT system calls 489
col 491
Colburn, Mark H. 52
com 491
com1 493
com2 494
com3 494
com4 495
comm 495
command
  definition 6
command, definition 685, 904
commands 495
  background 318
  COHERENT 317
  concurrent execution 318
  first part 6
  in files 319
  value 328
comment 101
compare strings 940, 948
compare two regions 758
compare two strings 939
compiler 93
  C 45
  error messages 1044
  function-calling conventions 419
  naming conventions 419
compiling without linking 96
compress 501
computer
  connecting via serial port 976
computer language 98
computer time accounting 52
con.h 502
conditional inclusion of code, end 383
conforming translator, mark 386
connector
  DB-9P 875
  RS-232C 875
  serial 875
console 502
const 506
const.h 506
cont 4
continue 506-507
control key 4
conv 507
convert string to floating-point number . . 949
convert string to long integer 951
convert string to unsigned long integer . . 952
cooked devices 957
cooked files 26
cooked terminal 977
copy a region of memory 756, 759
copy header into program 384
core 507
core dump file format 507
cos() 508
cosh() 508
cp 16, 509
cpdir 510
cpio 52, 511
cpp 94, 455, 513
  error messages 1045
crackers 4
creat() 514
cron 28, 41, 514
crontab 514
crypt 49, 516
crypt() 516
ct 517
ctags 517
ctime() 518
ctrl key 4
ctype 519
ctype.h 520
current directory 12, 27
current line within source file 386
curses 521
curses.h 529
cut 529

CWD 530

D

daemon 531
data formats 531
data structure 103
data types 531
date 39, 532
date of translation 385
db 44, 46-47, 533
  setting registers 535
DB-25 connector 875
DB-9P connector 875
dc 537
dcheck 538
dd 539
debugging 46
default 540
  directory 327
  prompt 327
definitions 540
deftty.h 541
del key 5
dereferencing, pointer 839
deroff 541
detab 542
device
  boot 444
  root 444
device drivers 542
device-independent I/O 25
df 19, 543
diff 544
diff3 545
dir.h 545
directory 9-10, 546
  current 12, 27
  home 10-11, 320, 327
  parent 327
  removing 18
  root 12, 27
dirent.h 546
dirs 546
disable 547
disk
  block 19
disk usage 19
disk, floppy 610
disk, MS-DOS 610
div() 547
div_t 547
division, integer 547, 712
do 330, 548
dollar 450
done 330
dos 548
dot command 328
double 550
drvld 27, 551
du 19, 551
dump 26, 51, 552
dumpdate 553
dumpdir 553
dumptape.h 554
dup() 554
dup2() 554

E

ebcdic.h 556
echo 320-321, 556
ed 5, 44, 556
  $ 124
  & 145
  * 124
  + ... 131
  - 131
  . (dot) 121, 130
  .= 125
  : 149
  <ctrl-D> 118
  = ... 124
  ? 134
  adding lines 121
  advanced commands 141
  backslash 136
  caret 144
  carriage return 118
  changing lines 126
  characters, special 129, 143
  commands, advanced 130, 141
  commands, global 150
  copying blocks of texts 134
  current, line 122, 148
  deleting lines 125
  file, editing commands 141
  file, name, in ed command 122
  global substitute 129
  global, command 137, 150
  inserting lines 122
  joining lines 137
  line, locators 134
  line, number 120
  line, number ranges 123
  line, number zero 133
  line, numbers, relative 130
  move, blocks of text 132
  pattern 127
  print command 123
  prompt character 120
  removing lines 125
  reverse searching 141
  sed 121
  special characters 143
  spliting lines 138
  substitute command 127
  tutorial 117
EDITOR 560
egrep 561

elif . . . 332
else . . . 111, 332, 562
elvis . . . 562
enable . . . 570
end . . . 571
end conditional inclusion of code . . . 383
end-of-file indicator . . . 574
endgrent() . . . 571
endpwent() . . . 571
enter . . . 4
enum . . . 572
ENV . . . 572
environ . . . 573
environmental variables . . . 573
envp . . . 574
EOF . . . 574
eol . . . 4
epson . . . 575
Epson MX-80 . . . 846
erase . . . 40
errno . . . 575
errno.h . . . 576
error message, return text of . . . 941
error messages
  assembler . . . 1045
  cc . . . 1044
  cc0 . . . 1048
  cc1 . . . 1057
  cc2 . . . 1058
  cpp . . . 1045
  fsck . . . 1059
  ld . . . 1058
  make . . . 1064
  nroff . . . 1065
  system . . . 1043
esac . . . 334
etext . . . 578
eval . . . 579
event scheduling . . . 41
ex . . . 579
example . . . 378
exec . . . 580
execl() . . . 580
execle() . . . 580
execlp() . . . 581
executable file . . . 581
executable files . . . 95
executable program . . . 95
execution . . . 581
execv() . . . 582
execve() . . . 582
execvp() . . . 584
exit . . . 108, 584
exit() . . . 584
exp() . . . 585
export . . . 327, 586
expr . . . 586
extern . . . 588

F

fabs() . . . 589
factor . . . 589
factor.c . . . 96
failure . . . 328
false . . . 332, 589
fblk.h . . . 589
fc . . . 589
FCEDIT . . . 590
fclose() . . . 590
fcntl() . . . 590
fcntl.h . . . 591
fd . . . 591
fd.h . . . 592
fdformat . . . 592
fdioctl.h . . . 593
fdisk . . . 593
fdisk.h . . . 595
fdopen() . . . 595
feof() . . . 596
ferror() . . . 596
fflush() . . . 598
fgetc() . . . 599
fgets . . . 104
fgets() . . . 600
fgetw() . . . 601
fi . . . 332
field . . . 601
FILE . . . 103, 602
file . . . 9, 602
  block special . . . 26
  concatenation . . . 14
  copying . . . 16
  creating empty . . . 321
  creation . . . 15
  include . . . 46
  links . . . 19
  modification time . . . 46
  move . . . 16
  name . . . 10
  of commands . . . 319
  protection . . . 48
  prototype . . . 23
  raw . . . 26
  removal of . . . 18
  rename . . . 16
  restoring . . . 51
  special character . . . 26
file descriptor . . . 603
file format
  archive file . . . 401
  COHERENT file . . . 545
  core dump . . . 507
file format, processing accounting . . . 392
file formats . . . 603
file locking, UUCP . . . 338, 1023
file system
  layout . . . 26
  root . . . 25
file, generate name for temporary file . . . 987

file, indicate end of 574
file, source, include 384
file, transfer to/from MS-DOS 548
file-creation mask 1003
fileno() 604
files
  backup 511, 552, 966, 1010
  cooked 26
fill an area with a character 761
filsys.h 605
filter 30, 605
find 605
find one string within another 948
fixstack 606
flexible arrays 403
float 607
floating-point number, create from string . 949
floating-point numbers 95
floor() 610
floppy disk
  bootable 442
floppy disk, copy MS-DOS files to/from . . 548
floppy disks 610
fnkey 613
font, soft 846
fopen 103, 106-107
fopen() 614
for 104, 108-109, 330, 615-616
fork() 616
fortune 617
fperr.h 617
fprintf() 617
fputc() 618
fputs() 618
fputw() 619
fread() 619
free() 619
freelist 1043
freopen() 620
frexp() 621
from 622
fscanf() 622
fsck 25, 623
  error messages 1059
fseek() 625
fstat() 626
ftell() 628
ftime() 628
function 99, 628
function keys 613
function, pointer to 839
fwrite() 629
fwtable 629

G

gallows 276
gcd() 631
general functions 631
generate name for temporary file 987
getc() 632
getchar() 633
getegid() 634
getenv() 634
geteuid() 635
getgid() 635
getgrent() 635
getgrgid() 636
getgrnam() 636
getlogin() 636
getopt() 637
getopts 639
getpass() 640
getpid() 640
getpw() 640
getpwent() 640
getpwnam() 641
getpwuid() 641
gets() 642
getty 643
getuid() 644
getw() 644
getwd() 645
GMT 39, 645
gmtime() 645
goto 646
grave accent 327
grep 38-39, 647
group 648
  id 47
  name 47
group structure 649
grp.h 649
gtty() 649
guillotine 276

H

hard disk 650
  adding another 652
hard disk, copy MS-DOS files to/from . . . 548
hash 653
hashing, example 952
hdioctl.h 653
head 653
header file 100, 104
header files 653
header, copy into program 384
help 7, 655
here document 685, 904
Hewlett-Packard LaserJet 846
high-level language 98
HOME 327, 656
home directory 10-11, 320
hp 656
hpd 657
hpr 657
hpskip 658
hs 658
hypot() 660

I

i-node 661
  list 25
I/O redirection 14, 29
i8086
  registers 420
i8087 417
icheck 661
identifier, define as macro 381
idle 444
if 332, 662
IFS 662
include code conditionally 382-384
include file 653
include source file 384
inclusion of code, conditional, end 383
index() 663, 939
init 444, 663
initialization 403, 664
initialization of pointers 839
ino.h 667
inode.h 667
install 667
instruction set 98
instructions 98
int 105, 669
integer division 547, 712
interrupt 669
introduction to C programming 97
io.h 669
ioctl() 669
ipc.h 670
isalnum() 670
isalpha() 670
isascii() 671
isatty() 671
iscntrl() 671
isdigit() 671
islower() 672
ispos() 672
isprint() 672
ispunct() 672
isspace() 673
isupper() 673
itom() 673

J

j0() 674
j1() 675
jn() 675
jobs 675
join 676

K

kermit 677
kermit, interactive 482
Kernighan, Brian W. 99
keyboard tables 680
keys, function 613
keyword
  parameters 325
kill 40, 43, 685
kill() 684
King David 238
King Lear 237
Korn shell 685
ksh 685
KSH_VERSION 703

L

L-devices 704
l.out 405
l.out.h 704
L.sys 705
l3tol() 708
LaserJet 846
LASTERROR 709
lc 6, 12, 709
LCK files 338, 1023
ld 45, 710
  error messages 1058
ldexp() 712
ldiv() 712
ldiv_t 712
let 713
lex 713
  $ 164
  %% 155
  %S 166
  %{ %} 172
  (and) 163
  * 161
  + ... 161
  164
  // 164
  < > 166
  ? 162
  abbreviations 165
  action 155
  alternatives 163
  angle brackets 166
  BEGIN action 166
  beginning of line $ 164
  braces 157
  braces, in patterns 162
  character classes 160
  context match 164-165
  context, separate 167
  context, start 166
  context, switch 168
  definitions 155, 166
  definitions section 170
  dot 159
  ECHO 169
  end of line 164

exception 160
grouping, () 163
header section 171
lex specification 153
macro 165-166
match, exception. 160
match, in context 164-165
match, longest 161
match, non-graphic characters 163
match, optional 162
non-graphic character 163
non-graphic characters 163
optional match 162
pattern 155
patterns 158-159
program generator. 153
regular expressions 158
REJECT 169
repetion, zero or more. 161
repetition 161
repetition, specific count 162
repetitions, zero or more 157
repetitions, zero or one 162
rules 155
rules, context start 166
rules, with same action. 158
section, header. 171
sections, definitions 170
start condition 166
statements 156
statements multiple 157
tokens 172
tutorial 153
yacc 172
yylex 170
yytext 158
yywrap. 156, 172
| 163
Lexicon 715
introduction 377
libm 96
libraries. 717
library. 99
C 45
limits.h 717
line control. 385
line numbering, reset. 385
linefeed 4
lines. 719
link() 719
linked list, example. 952
linker 95
error messages 1058
linker-defined symbols 720
linking without compiling 96
links. 19
ln 19, 720
load-module execution 581
locale-specific string transformation. . . . 955
localtime() 721
lock files 338, 1023
log() 722
log10() 723
logging in
definition 3
logging out 9
login. 9, 48, 723
time 53
login accounting 389
logmsg 724
long 724
long integer, create from string 951
longjmp() 724
look 725
loop 104
lower case
in file names 10
lp 725
lpd. 726
lpioctl.h. 726
lpr 727
lpskip 727
ls 6, 12, 728
lseek(). 729
ltol3() 730
lvalue 730

M

m4. . . . . 44, 46, 732
argument 177
argument substitution 177
changequote 178
decision-making macro 178
decr 181
define 176
divert 179
divnum 179
dnl 178
dumpdef. 177, 183
endless loop 184
errprint 179
eval. 181
expression evaluation 181
extra newlines 178-179
ifdef 178
ifelse 181
include 178
incr. 181
index. 181
macro name recognition 176
maketemp. 182
nestable quotes 176
output stream 179
quote marks removing 176
quoted text 175
repeat 181
sinclude 178
string length 181
substr 179

syscmd 182-183
translit 180
tutorial 175
undefine. 178
undivert 179
unquoted text 175
machine instructions. 46
machine.h 734
macro 44, 100, 734
macro, undefine. 385
madd() 734
mail 30, 36-37, 735-736
receiving. 37
main 45, 100, 102
main() 738
major device number 543
major number 739
make 46, 739
$*. 192
$< 192
$? 193
$@ 193
- 190, 196
.DEFAULT. 195
.IGNORE 195
.SILENT 195
.SUFFIXES 192
/usr/lib/makeactions 191-192
/usr/lib/makemacros 191-192
actions. 191-192
archive. 193-194
assembler 193
colon. 187, 193
command line 187, 190-192
command line, macro definition. 191
command line, options 190
command line, target specification . . . 192
command, error 191, 196
command, printing 191
comment 188
debug option 190
default rules 192
double colon 193
error messages 1064
error status 191, 196
errors 196
exit status. 196
file 187, 191
file modification time 191
file option 191
hyphen 190
ignore errors option 191, 196
interrupt 196
lex 193
macro 191
macro, definition. 188, 191
macro, printing. 191
macros. 191-192
Makefile 191
modification time 191
no execution option 191
no rules option 191
options 190
print option 191
printing 191
program, maintenance 194
program, specification 187, 191
return value 196
rules option. 191
silent option 191, 195
special targets 195
specification 187, 191
target 192, 195
target, line 193
target, printing. 191
target, program. 192
target, specification 192
test suites. 194
touch option 191
tutorial 185
usr/lib/makeactions 192
yacc 193
malloc() 742
malloc.h 744
man 7, 28, 744-745
Mandrake the Magician 872
manifest constant. 381, 746
manual
how to use 1
user reaction report 2
mark a conforming translator. 386
mask, default 1003
math.h 746
mathematics library 96, 746
mboot. 747
mcmp() 748
mcopy() 748
mdata.h 748
mdiv() 748
me 31, 749
tutorial 197
mem. 756
memccpy() 756
memchr() 757
memcmp() 758
memcpy() 759
memmove() 759
memok() 760
memory allocation 760
memory, copy 756, 759
memset() 761
mesg 761
message of the day 780
MicroEMACS. 31, 749
<ctrl-@> 205
<ctrl-A> 200
<ctrl-B> 200
<ctrl-C> 228
<ctrl-D> 203
<ctrl-E> 200

<ctrl-G> 211
<ctrl-L> 207
<ctrl-N> 200
<ctrl-P> 201
<ctrl-T> 207
<ctrl-U> 214
<ctrl-U><ctrl-L> 207
<ctrl-V> 201
<ctrl-W> 205
<ctrl-X> 212, 230
<ctrl-X>! 228
<ctrl-X>1 . . . . . 218, 220
<ctrl-X>2 220
<ctrl-X>< 229
<ctrl-X><ctrl-B> 218
<ctrl-X><ctrl-C> 202, 204
<ctrl-X><ctrl-F> 216
<ctrl-X><ctrl-N> 221
<ctrl-X><ctrl-P> 221
<ctrl-X><ctrl-R> 216
<ctrl-X><ctrl-S> 202
<ctrl-X><ctrl-V> 217
<ctrl-X><ctrl-W> 212, 216
<ctrl-X><ctrl-Z> 221
<ctrl-X>> 229
<ctrl-X>B 222
<ctrl-X>E 224
<ctrl-X>F 208
<ctrl-X>K 219
<ctrl-X>N 220
<ctrl-X>P 220
<ctrl-X>Z 221
<ctrl-Y> 204
<ctrl-Z> 213
<ctrl> 197
<del> 203
<esc>! 222
<esc>% 211
<esc>2 230
<esc>< 201
<esc><del> 203
<esc>> 201
<esc>? 230
<esc>B 200
<esc>C 206
<esc>D 203
<esc>F 200
<esc>L 206
<esc>R 210
<esc>S 209
<esc>U 206
<return> 200, 210
arguments 214
arguments, default value 214
arguments, deleting 215
arguments, increasing or decreasing . . 214
arguments, selecting values 215
arguments, with create window 220
arrow keys 200
automatic mode 229
back 200
backspace key 200
backward, end of line 200
backward, one space 200
backward, one word 200
beginning of text 201
block indentation 207
block-kill text 204
buffer status 218
buffer status command 218
buffer status command, with windows . 223
buffer status window 218
buffer, definition 215
buffer, delete 219
buffer, for killed text 204
buffer, how differs from file 215
buffer, naming 216
buffer, need unique names 218
buffer, prompting for new name 218
buffer, replace with named file 216
buffer, switch b 217
buffer, with windows 222
buffers, number allowed 218
cancel a command 211
capitalization 206
center line on screen 207
commands 209
compiling and debugging 229
copying text 223
cursor movement display 200
delete buffer command 219
delete key 203
delete text, versus killing 202
end macro command 224
end of text 201
erase text 202
erase text, by line 204
erase text, erasing spaces 202
erase text, to the left 203
erase text, to the right 203
execute macro command 224
exit 212
extended commands 212
f option 754
file and buffer commands 216
file, definition 215
file, how differs from buffer 215
file, naming 216
file, rename 216
file, replace buffer with named f 216
file, with windows 222
forward, end of line 200
forward, one space 200
forward, one word 200
help window 230
help, in MicroEMACS 230
kill and move commands 204
kill text, block 205
kill text, versus deleting 202

killing and deleting 202
left 200
line position 200
lowercase 206
metakey 227
move text 204
movement commands 200
next error 229
next line 200
number of buffers allowed 218
previous error 229
previous line 201
program interrupt 228
quit 202
quit without saving text 204
redraw screen 207
rename file 216
repetition 201
replace buffer with named file 216
restore (yank back) killed text 204
return indent 207
reverse search 210
right 200
saving text 202, 212
screen down 201
screen redraw 207
screen up 201
scroll down 221
scroll up 221
search and replace 211
search, forward 209
search, reverse 210
searching 209
store command 213
switch buffer command 222
switch buffers 217
text, block kill 205
text, capitalize 206
text, erase to left 203
text, erase to right 203
text, kill by lines 204
text, lowercase 206
text, move 204
text, move from one buffer to another . 217
text, restore (yank back) 204
text, saving 212
text, uppercase 206
text, write to new file 212
text, yank back (restore) 204
transpose characters 207
tutorial 197
uppercase 206
visit command 217
window manipulation 220
window, buffer status command use . . 223
window, copying text among 223
window, enlarge 221
window, move within 222
window, moving text among 223
window, number possible 220
window, saving text 223
window, scroll down 221
window, shifting between 220
window, shrink 221
window, use with editing 222
window, using multiple buffers 222
word wrap 208
write text to new file 216
yank back text 204, 215
MicroKVETCH Electronic Nag 247
microprocessor 98
min() 762
minit() 762
minor device number 543
minor number 762
mintfr() 762
misc 763
mitom() 763
mkdir 14, 763
mkfnames 764
mkfs 23, 764
mknod 768
mknod() 767
mktemp() 768
mneg() 768
mnttab.h 769
mode 13
mode field 21
modem 769
cabling 875
modem control 775
modem problems, UUCP 1016
modem, Trailblazer, with UUCP 1015
modemcap 773
modeminit 776
modf() 776
modulus 777
mon.h 777
more 14, 777
motd 780
mount 23, 781
mount() 780
mount.all 781
mount.h 782
mout() 782
move files 16
mprec.h 782
ms 28, 782
MS-DOS 784
differences from COHERENT 784
equivalent COHERENT commands . . . 784
on same hard drive as COHERENT . . . 594
MS-DOS, copies files to/from 548
MS-DOS, reading floppy 611
msg 35, 789-790
msg.h 790
msgctl() 790
msgget() 792
msgrcv() 793
msgs 38, 794

msgsnd() 795
msig.h 797
msqrt() 797
msub() 797
mtab.h 797
mtioctl.h 798
mtoi() 798
mtos() 798
mtype() 799
mtype.h. 799
mult() 799
multi-tasking, definition 784
multi-user, definition. 784
multiple source files 96
multiple-precision mathematics 799
multiprocessing execution. 685, 904
multiuser mode 445
mv. 16, 803
mvfree() 803

N

n.out.h 804
name of system 1020
name, generate for temporary file 987
named pipe 804
ncheck 804
newgrp 805
newline
  in C strings 45
newusr 48, 805
nkb 806
nlist() 808
nm. 809
not modifiable, type qualifier 506
notmem() 810
nptx. 811
nroff. 28, 33, 812
  % number register. 257
  %, page number 244
  .AB macro. 243
  .ad primitive 250-252
  .AE macro. 243
  .AI macro 243
  .AU macro. 243
  .BD macro 248
  .bp primitive 244, 248, 254
  .br primitive 250, 255
  .CD macro 248
  .ce primitive 253
  .da primitive 284
  .DE macro 247
  .di primitive. 283-284
  .DS macro. 247
  .ds primitive 244, 264
  .el primitive. 273
  .ev primitive 277
  .FE macro. 247
  .fi primitive 250, 252
  .FO macro. 257
  .FS macro. 247
  .ft primitive 282
  .hd primitive 257
  .ID macro 248
  .ie primitive. 273
  .IP macro 237
  .KE macro. 248
  .KS macro. 248
  .LD macro. 248
  .ll primitive 249, 271
  .ls primitive. 278
  .lt primitive 258, 281
  .na primitive 250, 252
  .nf primitive. 250
  .NH macro 241
  .nr primitive 266
  .pl primitive. 254
  .po primitive 249, 272, 277
  .PP macro 234, 236, 256
  .QE macro 241
  .QS macro. 241
  .RE macro. 238
  .RS macro. 238
  .SH macro. 241
  .sp primitive 236, 248, 253, 255
  .ta primitive. 254
  .tc primitive. 254
  .ti primitive 255
  .TL macro 243
  .tl primitive 258
  .wh primitive 257
  /usr/lib/tmac 288
  adjust 250
  begin page 248
  block-centered display 247-248
  boldface 245
  break 250, 252
  breaking line 236
  centered display 247-248
  characters, special. 246
  command, argument 236
  command, break. 250
  command, conditional 273
  command, divert. 283
  command, environment. 277
  command, fill. 250
  command, line length. 249
  command, line space 278
  command, page offset. 249, 277
  command, title length. 258
  command, when 257
  comments. 250
  conditional input 273
  CT string 244
  display. 247
  display indented 247
  display, block-centered 247-248
  display, centered. 248
  display, indented. 248
  display, left 248

diversion 283
error messages 1065
expression 270
fill 250
fonts 245
footer 244, 257
footnote 247
header 244
headings, section 241
hyphenation 236
indentation, relative 238
indented display 248
indented, display 247
italic 245
justify 250
justify text 235
keep 247-248
left display 248
line, length 249
LT string 244
macro 255
macro definition 233
macro, arguments 259-260
macro, definition 260
macro, name 236
margin, right 235
margins 250
measurement 271
measurement, absolute 274
measurement, units 257, 271
ms macros 233
new page 248
no-fill 250
numbered heading 241
page number 244
page, break 248
page, offset 249
paragraph 236, 252
paragraph tag 237
paragraph, indented 237
paragraph, quoted 241
quoted paragraph 241
register, number 266
relative indent 238
Roman 245
RT string 244
section heading 241
skip lines 248
space, vertical 236
specification 249
stack, environment 279
string 264
string, within strings 265
strings 244
tab 254
tag on paragraph 237
title 244
traps 254, 259
tutorial 231
unit, default 271
units 271
nroff macros 28
NUL 819
NULL 107, 819
null 820
null pointer 839
number of arguments, variable 1028
nybble 820

O

object format 821
object module 94, 96
od 821
open() 821
operator 822
precedence 824
operator, stringize 379
operator, token-pasting 380
operators 99, 408
optimization 94
optimizer/object generator 94
option 6
options 13
order
of matched file names 322
ospeed 978
output formatting 45

P

packed decimal 609
PAGER 825
param.h 825
parameter 6
assigning keyword 326
fewer 323
keyword 325
name 6
null 323
option 6
positional 322-323, 326
references 323
substitution 46, 329
parent directory 327
parentheses 45
Parkinson's law 19
parser 94
partition
root, changing size of 652
partition, adding COHERENT p. 650
passwd 28, 39, 48, 825-826
password 31, 39, 47-48
paste 826
patch 827
PATH 27, 327, 829
path name 11-12
fully specified 11
path() 829
path.h 830

paths 830
pattern 831
patterns 38-39, 320, 322
pause() 832
pax 832
PC 978
PCL 846
pclose() 832
permission
  access 20
  read 22
  write 21
Permissions 833
perror() 835
phone 835
PID 42
pinout
  DB-9P 875
  RS-232 875
pipe 30, 838
pipe() 836
pipeline, definition 685, 904
pnmatch() 838
pointer 99, 839
pointer dereferencing 839
pointer type 839
pointer type derivation 839
pointer-type mismatch 839
poll.h 842
popd 842
popen() 842
port 842
portability 843
PostScript 846
pow() 843-844
powering down 51
pr 33, 844
precedence 824
prep 845
preprocessing directive, include source file 384
preprocessing directive, reset line number 385
print 845
printer 575, 656-657, 727, 846
  cabling, serial 875
printf 45, 100, 104
printf() 848
proc.h 851
process 42, 318, 851
  background 318
  id 42, 318, 640
process accounting, file format 392
prof 851
profile 851
program
  debugging 46
  modularity 46
program execution 581
programming
  structured 46
prompt 31, 320, 327
protection 48
prototype 23
prps 852
prvd 407
prvi 407
ps 42, 318, 854
PS1 . . . . . 327, 855
PS2 . . . . . 327, 855
ptrace() 856
Puddnhead Wilson 256
pun 857
pushd 857
putc() 858
putchar() 858
puts() 859
putw() 859
pwd 15, 860
pwd.h 860

Q

qfind 861
qsort() 861
question mark 321
quot 862

R

RAM 865
ram 863
ram1 864
rand() 864
random access 865
ranlib 865
raw devices 958
raw files 26
raw terminal 977
rc 866
read 866
read permission 22
read() 866
read-only memory 868
readonly 867
real time 54
realloc() 868
reboot 868
rebooting COHERENT 7
receiving mail 37
redirection 14
ref 868
referenced type 839
region of memory, copy 756, 759
region of memory, search for character . . 757
regions, compare 758
register 98, 869
register declaration 869
register names 405
register variable 869
regular expressions 38
releases, software, preparing 667

remacc 825
remote access password 825
remote communication. 769
remove a directory 18
rename 869
rename a file 869
rename files 16
reset line number 385
restor 26, 870
restore
  files 51
return 4, 872
rev. 872
reverse search for character in string . . . 947
rewind(). 873
rindex() 873, 947
Ritchie, Dennis 93, 98
rm 18-19, 874
rmdir 18, 875
ROM 868
root 11-12, 30, 43, 875
  device 444
  directory. 27
  file system 25
root partition
  changing size of 652
rpow() 875
RS-232 875
rub out key 5
run time, check assertion 427
rvalue 877

S

sa 878
sbrk() 879
scanf(). 879
scat 881
sched.h 883
screen editor 31
script 317, 319, 323
SCSI. 883
sdiv() 883
search an array 447
search for character in a string 939, 947
search for character in region of memory . 757
search string for character 947
SECONDS 884
security. 884
sed 885
  $ 295
  > 292
  change lines 301
  ed 291
  including a file 302
  line range 295
  line selection 294
  next line. 303
  p command withs 297
  pattern 295
  pipes 291
  reading in 302
  substitution 293
  tutorial 291
seg.h 887
sem 888
sem.h 888
semctl() 889
semget(). 890
semicolon 100
semicolons 317
semop() 891
set 893
setbuf() 895
setgid() 895
setgrent() 896
setjmp(). 896
setjmp.h 897
setpwent() 897
settz() 898
setuid() 898
sgtty. 898
sgtty.h 904
sh 27, 29, 319, 904, 317
SHELL 914
shell. 27, 29, 317
  script 319
  sequential execution of commands . . . 317
  simple commands 317
  variable 324, 327
shell, Bourne 904
shell, Korn 685
shellsort(). 914
shift 915
shm 915
shm.h. 916
shmctl(). 917
shmget() 918
short 919
shrd 407
shri 407
shutdown 919
shutting down COHERENT 7
signal() 919
signal.h. 921
signame 921
sin() 921
single-user mode 445
sinh() 921
size 922
sizeof 922
slash
  in path name 11
sleep 923
sleep(). 923
sload(). 923
smult() 924
soft fonts 846
software, installing under COHERENT . . 667
software, preparing releases. 667

sort 924
source file 96
source file inclusion 384
source file name. 386
source file, current line 386
source file, time translated 387
space 324
special file
  block. 26
spell. 925
split 926
spow(). 926
sprintf(). 927
sqrt() 927
srand() 927
srcpath 928
ss 928
sscanf() 930
stack 931
stack size. 97
stackalter size of 712
standard
  input. 30
  output 29
standard error. 931
standard I/O. 28
standard input 931
standard output. 14, 932
stat() 932
stat.h 934
static 934
stdarg.h. 934
stddef.h. 935
stderr 935
stdin 935
STDIO. 935
stdio.h 104, 936
stdlib.h 936
stdout. 937
sticky bit 480, 937
stime() 938
storage class 938
strcat() 938
strchr() 939
strcmp(). 939
strcoll() 939
strcpy() 940
strcspn() 940
stream 940
stream.h 941
strerror() 941
string functions 942
string transformation, locale specific . . . 955
string, break into tokens. 949-950
string, compare two. 939
string, comparison 940, 948
string, convert to floating-point number . 949
string, convert to long integer. 951
string, convert to unsigned long integer. . 952
string, find one within another 948
string, reverse search for character 947
string, search for character 947
string, search for character in 939, 947
string-ize operator 379
string.h 941
strings 944
strip 944
strlen() 944
strn 407
strncat() 945
strncmp() 945
strncpy() 945
strpbrk() 947
strrchr(). 947
strspn() 948
strstr() 948
strtod() 949
strtok() 949-950
strtol(). 951
strtoul() 952
struct 954
structure 955
  group 649
structure assignment. 955
structured
  programming. 46
structured programming. 99
strxfrm() 955
stty 5, 40, 957
stty(). 956
su 43, 959
subject sequence 949, 951-952
substitution
  in commands. 320
  of parameters. 46, 329
success 328
succotash 514
suload() 959
sum 959
superuser 30, 43, 960
swab() 960
switch. 961
symt. 407
sync. 25-26, 445, 962
sync() 961
system
  time 54
system calls 489
system maintenance 962
system name. 1020
system(). 962

T

tab. 40
tail. 964
tan() 964
tanh() 964
tape 965
tar 49, 966

technical information . . . 968
tee . . . 968
tempnam() . . . 969
temporary file, generate name . . . 987
TERM . . . 969
termcap . . . 969
terminal . . . 40, 976
  cabling . . . 875
  cooked . . . 977
  functions . . . 898
  interface . . . 898
  mode . . . 26
  raw . . . 977
terminal-independent operations . . . 978
termio.h . . . 979
test . . . 329, 979
testing
  strings . . . 329
text of error message, return . . . 941
tgetent() . . . 981
tgetflag() . . . 981
tgetnum() . . . 981
tgetstr() . . . 982
tgoto() . . . 982
The C Programming Language . . . 99
Thompson, Ken . . . 93
time . . . 39, 983
time source file is translated . . . 387
time() . . . 982
time.h . . . 984
timeb.h . . . 984
timef.h . . . 984
timeout.h . . . 984
times . . . 985
times() . . . 984
times.h . . . 985
TIMEZONE . . . 985
timezone . . . 39
tmpnam() . . . 987
token pasting . . . 380
token, break a string into sequence of 949-950
token, definition . . . 685, 904
token-pasting operator . . . 380
tolower() . . . 987
touch . . . 988
toupper() . . . 988
tputs() . . . 989
tr . . . 989
Trailblazer modem with UUCP . . . 1015
transform a string . . . 955
translation, date . . . 385
translator, mark conforming . . . 386
trap . . . 990
troff . . . 33, 991
true . . . 996
tsort . . . 996
tty . . . 996
tty.h . . . 997
ttyname() . . . 997
ttys . . . 997
ttyslot() . . . 999
ttystat . . . 999
type checking . . . 1000
type promotion . . . 1000
type qualifier, not modifiable . . . 506
type, pointer . . . 839
type, referenced . . . 839
typedef . . . 1000
types.h . . . 1000
typeset . . . 1001
typo . . . 1001

U

umask . . . 1003
umask() . . . 1003
umount . . . 24-25, 1004
umount() . . . 1004
unalias . . . 1004
uncompress . . . 1005
undefine a macro . . . 385
ungetc() . . . 1005
union . . . 1005
uniq . . . 1006
units . . . 28, 1006
unlink() . . . 1008
unmkfs . . . 1008
unsigned . . . 1009
unsigned long integer, create from string . 952
until . . . 333, 1009
update . . . 445, 1009
uproc.h . . . 1010
USENIX . . . 52
USER . . . 1010
user
  id . . . 47
  name . . . 47
  time . . . 54
user name . . . 29
user reaction report . . . 2
usr . . . 11
ustar . . . 49, 1010
utime() . . . 1011
utmp.h . . . 1012
utsname.h . . . 1012
uucheck . . . 1012
uucico . . . 1013
UUCP . . . 1014
uucp . . . 1019
UUCP
  domain name . . . 1018
  lock files . . . 338, 1023
  system name . . . 1018
  tutorial . . . 335
uucpname . . . 1020
uudecode . . . 1020
uuencode . . . 1020
uuinstall . . . 1021
uulog . . . 1022
uumvlog . . . 1022

uuname 1022
uurmlock 1023
uutouch 1023
uux 1024
uuxqt 1026

V

v7sgtty.h 1027
va_arg() 1027
va_end() 1027
va_start() 1028
value from command 328
variable
  shell 324, 327
variable arguments 1028
vertical bar 334
vi 1029
view 1030
virec 1030
void 1031
volatile 1031

W

wait 318, 1032
wait() 1032
wall 1033
wc 1033
whence 1033
whereis 1034
which 1035
while 333, 1035
who 29, 35, 50, 1036
wild pointer 839
wildcards 1036
word 440
write 35, 1037
write permission 21
write() 1037
wtmp (5c) 866

X

xgcd() 1039

Y

yacc 1040
  %% 356
  %left 365-367
  %nonassoc 367
  %prec 366
  %right 365-366
  %token 362-363, 365
  accept action 356
  action statements 357
  action, accept 356
  action, error 356
  action, reduce 356
  action, shift 356
  actions 358
  ambiguity 363
  ambiguity, default handling 364
  ambiguity, resolution 365
  associative, left 365
  associative, right 365
  associativity 365
  Backus-Naur Form 355
  BNF 355
  comments, in rules 358
  default, action 363
  definition section 356, 365
  definitions section 357
  error action 356
  error, recovery 367
  error, token 367
  LALR 355
  left-to-right parsing 355
  library 356
  library, yacc 356
  LR parsing 355
  nonassociative 367
  nonterminals 357
  parse actions 356
  precedence 365-366
  production 357
  push-down list 356
  reduce 356, 364
  reduction 357
  right 367
  rule format 358
  rule, actions 359
  rule, format 358
  rule, sections 357
  rule, style 358
  rule, type 363
  rule, values 360
  rules section 356
  rules, precedence 366
  section, definition 356
  section, rules 356
  shift 356, 364
  shift-reduce conflicts 364
  stack 356
  start symbol 356-357
  terminals 357
  token definition 357
  token, definition 362
  token, error 367
  token, value 360
  tutorial 351
  type, of nonterminal 363
  user code 357
  value, qualification 363
  yyerrok 367
  yyparse 356
  {} 357
yes 1041

Z

zcat 1042
zerop(). 1042

{} 45

| 30, 334
|| 329

User Evaluation Report

To keep this manual free of errors and to help us improve **COHERENT**, please sent us your reactions. Please fill in the form below, detach it, and mail it to:

Mark Williams Company
60 Revere Drive
Northbrook, IL 60062

Name: _____

Company: _____

Address: _____

City/State/Zip: _____

Phone: _____

Date: _____

Version and hardware used:

Did you find any errors in the manual?

Can you suggest any improvements to the manual?

Did you find any bugs in the software?

Can you suggest any improvements or enhancements to the software?

Additional comments:

CTRL \ - quit character
CTRL C - default interrupt character

QIC-24 tape Front

QIC-36 } interface specs
QIC-02 }

IDE drives controller function, are located

MARK WILLIAMS COMPANY ("MWC")
Software License Agreement

YOU SHOULD CAREFULLY READ THIS SOFTWARE LICENSE AGREEMENT BEFORE BREAKING THE SEAL ON THE DISKETTE ENVELOPE. BREAKING THE SEAL INDICATES YOUR ACCEPTANCE OF THE TERMS AND CONDITIONS OF THIS AGREEMENT. IF YOU DO NOT AGREE WITH THEM, YOU SHOULD PROMPTLY RETURN THE DISKETTE UNOPENED, AND YOUR MONEY WILL BE REFUNDED.

MWC provides this software and licenses its use to you. You assume responsibility for the selection of the software to achieve your intended results, and for the installation, use and results obtained from it.

LICENSE

MWC grants you a license only to: (a) use the software on a single machine; and (b) copy the software into any machine readable form for backup purposes in support of your use of the software on the single machine.

As an exception to the foregoing paragraph, we grant you the right to include portions of the MWC Runtime Library (as defined below) in software programs that you develop, called Composite Programs, and to use, distribute and license Composite Programs to third parties without payment of any fee. You shall, however, include in each Composite Program, and on the exterior label of every diskette, a copyright notice in this form: "Portions of this program, copyright 1982, 1990, Mark Williams Company." As an express condition to the use of the software, you agree to indemnify and hold MWC harmless from all claims by you and third parties arising out of the use of any Composite Program. Any portion of the Runtime Library merged into another program will continue to be subject to the terms and conditions of this Agreement. "Runtime Library" is defined as the set of copyrighted MWC language subroutines provided with the software, a portion of which must be linked to and become part of a Composite Program for that Program to run on a computer.

You may not transfer the software, or any copy, modification or merged portion, in whole or in part, except as expressly provided herein.

TERM

You may terminate the license at any time by destroying the software together with all copies, modifications and merged portions in any form. It will also terminate upon conditions set forth elsewhere in this Agreement or if you fail to comply with any terms or conditions of this Agreement. You agree upon termination to destroy the software together with all copies, modifications and merged portions in any form. The license shall terminate upon termination of this Agreement.

LIMITED WARRANTY

EXCEPT FOR THE LIMITED WARRANTY SET FORTH IN THE NEXT PARAGRAPH, THE PROGRAM IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU (AND NOT MWC OR ANY AUTHORIZED MWC DEALER) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. MWC DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE SOFTWARE WILL BE UNINTERRUPTED OR ERROR FREE.

MWC warrants to the original licensee that the diskettes on which the software program is recorded is free from defects in material and workmanship under normal use and service for a period of 60 days from the delivery date as evidenced by a copy of your receipt. Your exclusive remedy is the return of the diskettes as described below.

IN NO EVENT WILL MWC BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE EVEN IF MWC OR AN AUTHORIZED MWC DEALER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

MISCELLANEOUS

If you are dissatisfied with the diskette[s] for any reason whatsoever (including if the diskettes are defective in material or workmanship), you may return them for replacement or refund of amounts previously paid them so long as the return is made within 60 days after purchase, and is accompanied by (a) a Return Authorization Number form Mark Williams Company, (b) your receipt, (c) an affirmative statement that you have not retained any copies of the software (including back-up copies) and (d) a statement as to the reason for the return.

You may not sublicense, assign or transfer the license to the software except as expressly provided in this Agreement. Any attempt otherwise to sublicense, assign or transfer any of the rights, duties or obligations hereunder is void. This Agreement will be governed by the laws of the State of Illinois.

Should you have any questions concerning this Agreement, you may contact MWC by writing to the Mark Williams Company, 60 Revere Drive, Northbrook, IL 60062.

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU FURTHER AGREE THAT IT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN US WHICH SUPERSEDES ANY PROPOSAL OR PRIOR AGREEMENT, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN US RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

COHERENT™

Mark Williams
Company